CS Exit Ethiopia

Questions & Worked Answers

Computer Science Exit Examination

271 complete questions and 4 source variants

1 selected course | 18 November 2026

Selected courses

01. Data Structure and Algorithms

https://csexitethiopia.com

abelasnake2003@gmail.com | Telegram: @Abechu7

Course contents

Each course begins on a new page. PDF bookmarks provide question, study-note section and revision-topic navigation.

Data Structure and Algorithms 3

Reading the answers

Every question includes its answer, explanation, choice-by-choice contrasts and available evidence. Primary note proof quotes the study notes. Related reading and module passages are labelled separately; a module reference is not presented as primary note proof.

Data Structure and Algorithms

Questions & Worked Answers

271 questions + 4 source variants

Question 1 | Q00012

When we use a hash table data structure, our hash function should map the expected inputs (keys) as evenly as possible over its output range(indexes). This property of good hash function is known as_____.

A. Uniformity

B. Determinism

C. Continuity

D. Data normalization

Hint

Trace the initial state and each executed operation in order, preserving the requested output layout.

Answer A: Uniformity

DSA25 defines uniformity as spreading expected keys evenly among addresses. That directly matches the question's wording. Determinism is a different obligation: repeated equal keys receive the same address for the chosen function. Continuity and data normalization do not name this distribution property, so only Uniformity answers the literal stem.

Why each choice is right or wrong

A - Correct: Uniformity describes spreading expected input keys evenly across the output address range, exactly as the stem requires.

B - Incorrect: Determinism requires repeated equal keys to map identically; it does not require an even address distribution.

C - Incorrect: Continuity is not the key-distribution property defined for hash functions in DSA25.

D - Incorrect: Data normalization does not name the even-distribution requirement of a hash function.

Primary note proof

Study-note section DSA25

> Determinism means repeated equal keys get the same address for the chosen function. Uniformity means expected keys spread evenly. Universal hashing chooses one function randomly from a suitable family for the table, independently of a fixed key set; it does not choose a fresh unrelated function for each lookup. Random choice controls collision probability, not a promise of zero collisions.

Question sources

MOE Computer Science Exit Exam · 2016 · Question 12

Question 2 | Q00013

In a one-based array representation of a binary heap, the root is at index 1. If a node at index i has a right child, at which index is that child stored?

A. 2*i+1

B. i/2

C. (i+1)/2

D. 2*i

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. 2*i+1

Heap child formulas depend on whether indexing begins at zero or one. With the explicitly stated one-based convention, the two children of node i occupy positions 2i and 2i+1. The latter is the right child. Integer division by two instead identifies a parent in this convention. The existence condition matters because an index computed beyond the current heap size does not designate an actual child.

Why each choice is right or wrong

A - Correct: 2i+1 is the one-based right-child index.

B - Incorrect: floor(i/2) computes the parent for nonroot nodes.

C - Incorrect: (i+1)/2 does not compute the right child.

D - Incorrect: 2i is the one-based left child.

Primary note proof

Study-note section DSA23

> Zero-based array representation: root index 0, left=2i+1, right=2i+2, parent=floor((i-1)/2) for i>0. One-based representation uses children 2i,2i+1 and parent floor(i/2). A child exists only if its index is inside the current heap size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

> iii. Heap Sort
> Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
> each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
> Algorithm:

1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

MOE Computer Science Exit Exam · 2016 · Question 13

Question 3 | Q00014

Use iterative sequential search and iterative binary search on random-access arrays, with constant-size indices. For binary search, reverse comparisons on descending arrays. Compare tight asymptotic auxiliary space and worst-case element probes as n grows. Which statement is false?

A. Binary searching algorithm can work on list of data which is stored in descending order

B. Sequential and binary searching algorithm are not equal in terms of memory efficiency

C. Sequential searching algorithm can work on sorted and unsorted list of data

D. Binary search has a smaller tight worst-case probe order than sequential search on sorted random-access arrays.

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer B: B. Sequential and binary searching algorithm are not equal in terms of memory efficiency

Both iterative algorithms keep only a fixed number of indices and temporary values, giving constant auxiliary space. Their worst-case probe counts differ: sequential search can inspect every element, whereas binary search halves the ordered interval each time. Binary search can also handle descending order with reversed comparisons, and sequential search does not require sorted input. Under these specified implementations, the unequal-memory claim is the false statement.

Why each choice is right or wrong

A - Incorrect: Descending binary search works with reversed comparisons.

B - Correct: Both iterative implementations use constant auxiliary space, so this inequality claim is false.

C - Incorrect: A sequential scan works with either input ordering.

D - Incorrect: Binary search has logarithmic rather than linear worst-case probes.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Study-note section DSA29

> Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first element
> • Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

MOE Computer Science Exit Exam · 2016 · Question 14

Question 4 | Q00045

For ordinary array sorting implementations, which listed algorithm uses a temporary element buffer proportional to n rather than only a fixed number of saved elements? Count recursive call frames separately from auxiliary element storage.

A. Insertion sort

B. Selection sort

C. Merge sort

D. Quick sort

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: C. Merge sort

Ordinary array mergesort merges two sorted halves through a temporary buffer that can hold a number of elements proportional to n. Insertion sort retains one item while shifting, and selection sort needs fixed temporary storage for a minimum position and an exchange. Ordinary quicksort partitions inside the original array. Its recursive stack is counted separately here, so the linear element-buffer requirement uniquely identifies mergesort.

Why each choice is right or wrong

A - Incorrect: Insertion saves one item and shifts within the same array.

B - Incorrect: Selection retains a minimum position and a fixed-size swap temporary.

C - Correct: Ordinary array merging needs a linear auxiliary element buffer.

D - Incorrect: In-array partitioning needs no full merge buffer; recursive frames are excluded by the stated convention.

Primary note proof

Study-note section DSA37

> Ordinary array merging needs a linear auxiliary buffer.

Study-note section DSA34

> For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i].

Study-note section DSA36

> Quicksort normally rearranges in the same array but recursive storage can be linear in the worst case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

MOE Computer Science Exit Exam · 2016 · Question 45

Question 5 | Q00066

Which data structure is the most appropriate for sharing a printer in a network environment?

A. Stack

B. Binary search tree

C. Full binary tree

D. Queue

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: Queue

The module explicitly identifies print-server jobs as a queue application. FIFO preserves arrival order among pending jobs. Evidence: assigned PDF pages 8, 6, 9.

Why each choice is right or wrong

A - Incorrect: A stack would give the most recent print request priority. See assigned PDF 8, 6, 9.

B - Incorrect: BST ordering by key is not arrival-order service. See assigned PDF 8, 6, 9.

C - Incorrect: A full tree shape does not specify job service order. See assigned PDF 8, 6, 9.

D - Correct: The print server maintains a queue of print jobs. See assigned PDF 8, 6, 9.

Primary note proof

Study-note section DSA14

> FIFO suits equal-priority printer jobs, shared-resource waiting, asynchronous buffers and level-order exploration. A play sequence can also use a queue. Distinguish airline seat records from waiting requests; an application name alone does not force one representation.

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 41-48

> Application of Queue
> Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
> • Print server- maintains a queue of print jobs
> • Disk Driver- maintains a queue of disk input/output requests
> • Task scheduler in multiprocessing system- maintains priority queues of processes

• Telephone calls in a busy environment -maintains a queue of telephone calls
• Simulation of waiting line- models real life queues (e.g. supermarkets checkout, customer service line (Bank, custom, airport etc))

Data Structure and Algorithms module | PDF page 6 | lines 15-22

"A stack is an ordered list in which all insertions and deletions are made at one end, called the top". Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

MOE Computer Science Exit Exam · 2016 · Question 66

Question 6 | Q00076

Which one of the following is true about Queue data structure?

A. Queue data structure applies first come first served policy

B. Queue data structure uses last in first out rule

C. Queue data structure does not have efficient basic operations

D. Queue data structure cannot be implemented using linked list

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Queue data structure applies first come first served policy

FIFO serves the first arrival first; the module also explicitly gives linked-list enqueue and dequeue. Therefore A is true and the supplied D key contradicts page 8. Evidence: assigned PDF pages 7, 8.

Why each choice is right or wrong

A - Correct: First come first served is stated under queue applications. See assigned PDF 7, 8, 8.

B - Incorrect: LIFO describes the stack, whereas queue deletion is at the front. See assigned PDF 7, 8, 8.

C - Incorrect: A blanket inefficiency claim is unsupported; front removal and rear insertion can use fixed pointer updates. See assigned PDF 7, 8, 8.

D - Incorrect: The module expressly implements the queue as a linked list. See assigned PDF 7, 8, 8.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 8 | lines 35-37

> III. Linked list implementation of enqueue and dequeue operations
> Enqueue- is inserting a node at the end of a linked list
> Dequeue- is deleting the first node in the list

Data Structure and Algorithms module | PDF page 8 | lines 41-48

> Application of Queue
> Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
> • Print server- maintains a queue of print jobs
> • Disk Driver- maintains a queue of disk input/output requests
> • Task scheduler in multiprocessing system- maintains priority queues of processes
> • Telephone calls in a busy environment -maintains a queue of telephone calls
> • Simulation of waiting line- models real life queues (e.g. supermarkets checkout, customer service line (Bank, custom, airport etc))

Question sources

MOE Computer Science Exit Exam · 2016 · Question 76

Question 7 | Q00148

An unsorted singly linked list maintains only its head pointer. Which option gives the complete set of listed operations with tight worst-case constant time? There is no cached tail, predecessor index or positional index.

A. Insertion at the front of the linked list

B. Insertion at the end of the linked list

C. Deletion of the front node of the linked list

D. Deletion of the last node of the linked list

E. A&C

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer E: E. A&C

Inserting at the head links the new node to the old head and updates head. Deleting the head advances head and releases one node. Both use constant work. Appending requires traversal to find the tail, and deleting the tail requires its predecessor, also found by traversal. The complete constant-time set is therefore insertion at the front together with deletion of the front, which is option E.

Why each choice is right or wrong

A - Incorrect: Front insertion is constant, but this alone omits the also-constant front deletion.

B - Incorrect: Head-only append traverses the list to find its tail.

C - Incorrect: Front deletion is constant, but this alone omits the also-constant front insertion.

D - Incorrect: Tail deletion must locate its predecessor by traversal.

E - Correct: This combines exactly the two listed constant-time operations.

Primary note proof

Study-note section DSA09

> Prepend with new.next=head; head=new. To insert after a known predecessor p, set new.next=p.next before p.next=new. Appending with only head requires a tail traversal; maintaining tail avoids it. Delete the head by saving old=head, advancing head=head.next and releasing old in C++.

Study-note section DSA09

> Deleting the tail of a singly linked list requires its predecessor, even when tail itself is known. Handle empty and single-node lists separately; after removing the only node, set both head and

tail to null. To remove a known successor, bypass it before deallocation. Clearing an owned n-node chain visits and releases every node.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 13-24

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.
if (start == NULL) // If list is empty, then new node become start node
start = temp;
else {
temp2 = start; // list not empty! Thus temp2 starts from first node
while (temp2->next!= NULL) { // till the current last node found navigates forward
temp2 = temp2->next; // Move to next link in chain
}
temp2->next = temp; // new node (temp) appended at the end of the list

Data Structure and Algorithms module | PDF page 6 | lines 4-13

2. Make temp1 point to whatever the start pointer is pointing to.
3. If the next pointer of what temp1 indicates is NULL, then we've found the last node of the list, so
jump to step 7.
4. Make another pointer, temp2, point to the current node in the list.
5. Make temp1 point to the next item in the list.
6. Go to step 3.
7. If you get this far, then the temporary pointer, temp1, should point to the last item in the list and
the other temporary pointer, temp2, should point to the last-but-one item.

8. Delete the node pointed to by temp1.
9. Mark the next pointer of the node pointed to by temp2 as NULL - it is the new last node.

Question sources

Addis Ababa University mock exam · Question 50

Question 8 | Q00150

Which of the following data structure is more appropriate to represent a heap tree?

A. Two dimensional array

B. Doubly linked list

C. Linear array

D. Single linked list

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Linear array

The heap algorithm directly maps tree nodes to Data indices and uses arithmetic child indices, establishing the linear array representation. Evidence: assigned PDF pages 11.

Why each choice is right or wrong

A - Incorrect: A second array dimension is unnecessary for one index per node. See assigned PDF 11.

B - Incorrect: A doubly linked list lacks the given direct child-index arithmetic. See assigned PDF 11.

C - Correct: Data[0] and children 2*i+1, 2*i+2 use a one-dimensional array. See assigned PDF 11.

D - Incorrect: A singly linked chain does not directly represent the two children by these indices. See assigned PDF 11.

Primary note proof

Study-note section DSA23

Zero-based array representation: root index 0, left=2i+1, right=2i+2, parent=floor((i-1)/2) for i>0. One-based representation uses children 2i,2i+1 and parent floor(i/2). A child exists only if its index is inside the current heap size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Addis Ababa University mock exam · Question 52

Question 9 | Q00151

Linear search is highly inefficient compare to binary search when dealing with_____?

A. Small, unsorted arrays

B. Large, sorted arrays

C. Small, sorted arrays

D. Large, unsorted arrays

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: Large, sorted arrays

The comparison n versus log2 n becomes substantial for large n, and binary searching requires ordering. Thus both size and sortedness matter. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: Unsorted input does not satisfy binary-search prerequisites. See assigned PDF 3, 3.

B - Correct: Large sorted arrays provide both the prerequisite and the asymptotic advantage. See assigned PDF 3, 3.

C - Incorrect: Sorting permits binary search, but small size does not produce the large growth-rate difference requested. See assigned PDF 3, 3.

D - Incorrect: Large unsorted arrays still cannot be binary-searched without preprocessing. See assigned PDF 3, 3.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Study-note section DSA29

> Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first element
> • Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

Addis Ababa University mock exam · Question 53

Question 10 | Q00152

Considering the array elements (independently for this question):
int a[6]={11,13,15,17,19,21}
Implementing stack, the elements after pop(); push(9); are:

A. 9, 11, 13, 15, 17, 19

B. 13, 15, 17, 19, 21, 9

C. 11, 13, 15, 17, 19, 9

D. 9, 13, 15, 17, 19, 21

Hint

Trace the initial state and each executed operation in order, preserving the requested output layout.

Answer C: 11, 13, 15, 17, 19, 9

Take the listed array in bottom-to-top order, with 21 at top. Pop removes 21; push appends 9 at the same top slot, leaving 11,13,15,17,19,9. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: This puts 9 at the bottom and shifts surviving data. See assigned PDF 6, 6.

B - Incorrect: This removes 11, which is a queue-style front removal. See assigned PDF 6, 6.

C - Correct: Only the final top entry changes from 21 to 9. See assigned PDF 6, 6.

D - Incorrect: This replaces the bottom entry 11 rather than the top. See assigned PDF 6, 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 6 | lines 24-42

```
I. Array implementation (static memory).
Push(n): Adding item at top
top is the current top of stack and n is its
maximum size
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;
```

Question sources

Addis Ababa University mock exam · Question 54

Question 11 | Q00153

A capacity-six circular queue uses all six slots with front=0 and size=6. Its physical array is [11,13,15,17,19,21]. After dequeue() and enqueue(9), what is the logical front-to-rear sequence?

A. 9, 11, 13, 15, 17, 19

B. 13, 15, 17, 19, 21, 9

C. 11, 13, 15, 17, 19, 9

D. Queue overflow

Hint

Trace the initial state and each executed operation in order, preserving the requested output layout.

Answer B: B. 13, 15, 17, 19, 21, 9

The initial front element is 11, so dequeue removes it and advances front to the slot containing 13. The queue now has five items. Circular insertion reuses the freed physical slot zero for 9 and restores size six. Reading from the new front around to the rear gives 13,15,17,19,21,9. Physical slot order differs from this requested logical service order.

Why each choice is right or wrong

A - Incorrect: This would remove 21 and prepend 9 rather than remove the front.

B - Correct: This is the front-to-rear order after removing 11 and appending 9.

C - Incorrect: This incorrectly retains 11 and removes the rear 21.

D - Incorrect: Circular reuse makes room for the insertion after the removal.

Primary note proof

Study-note section DSA15

> A simple noncircular array queue may report overflow when rear reaches the final index despite released slots at the front. A circular queue reuses slots with modulo arithmetic. Use a count convention: front points to the next removal, size counts items, and insertion slot is (front+size)%C. Empty means size==0; full means size==C.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 7 | lines 18-23

> I. Simple Array implementation (static memory).
> A simple array structure can be used to implement enqueue & dequeue operations. Thus item will
> be added at the end of the array whereas dequeue operation removes items from the front. In simple
> array implementation, to add item it checks whether there is room at the rear or not. Therefore, even though there are free released spaces at the front ot the array the queue implementation might
> generate overflow message. This is the key drawback to use simple array to implement Queue.

Data Structure and Algorithms module | PDF page 7 | lines 24-26

> II. Circular array implementation of Queue Operations
> To overcome the above drawback of simple array implementation a circular array can be used, in
> which freed spaces are re-used to store data.

Question sources

Addis Ababa University mock exam · Question 55

Question 12 | Q00154

A circular queue has capacity six, front=0 and size=6, using physical slots [11,13,15,17,19,21]. It can use all six slots. After dequeue() and enqueue(9), list the physical array in index order 0 through 5.

A. 9, 11, 13, 15, 17, 19

B. 13, 15, 17, 19, 21, 9

C. 11, 13, 15, 17, 19, 9

D. 9, 13, 15, 17, 19, 21

Hint

Trace the initial state and each executed operation in order, preserving the requested output layout.

Answer D: D. 9, 13, 15, 17, 19, 21

Dequeue reads 11 at index zero, advances front to index one and reduces size to five. The count-based insertion formula then gives (front+size)%6=(1+5)%6=0. Enqueue therefore writes 9 into the freed slot zero. All other physical slots retain their old values. The logical service order begins at 13, but the requested physical index order begins at the newly stored 9.

Why each choice is right or wrong

A - Incorrect: This drops 21 and retains the removed 11.

B - Incorrect: This is logical front-to-rear order, not physical index order.

C - Incorrect: This overwrites 21 instead of reusing slot zero.

D - Correct: Slot zero receives 9 and slots 1-5 remain unchanged.

Primary note proof

Study-note section DSA15

> A simple noncircular array queue may report overflow when rear reaches the final index despite released slots at the front. A circular queue reuses slots with modulo arithmetic. Use a count convention: front points to the next removal, size counts items, and insertion slot is (front+size)%C. Empty means size==0; full means size==C.

Study-note section DSA15

> Dequeue reads A[front], sets front=(front+1)%C and decreases size. An alternative reserves one unused slot and uses next(rear)==front for full; its empty test and capacity differ. State the convention before tracing.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 7 | lines 24-26

> II. Circular array implementation of Queue Operations
> To overcome the above drawback of simple array implementation a circular array can be used, in
> which freed spaces are re-used to store data.

Question sources

Addis Ababa University mock exam · Question 56

Question 13 | Q00171

The data structure used in the standard implementation of Breadth First Search is ?

A. Stack

B. Array

C. Linked List

D. Queue

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: Queue

Breadth-first search repeatedly dequeues a vertex and enqueues its undiscovered neighbors. FIFO ordering ensures earlier distance layers are processed before later ones. Arrays or linked nodes can store this frontier, but Queue names its required access discipline. A stack instead serves latest pending work.

Why each choice is right or wrong

A - Incorrect: A stack serves the latest pending vertex first and does not preserve BFS distance-layer order.

B - Incorrect: An array can store a queue, but an array alone specifies no FIFO frontier discipline.

C - Incorrect: Linked nodes can represent a queue, but Linked List alone does not specify breadth-first processing.

D - Correct: A FIFO queue processes discovered vertices by distance layer, the standard BFS frontier rule.

Primary note proof

Study-note section DSA41

Mark the source when enqueuing it. Repeatedly dequeue a vertex, examine its neighbours, and enqueue each unvisited neighbour while marking it. Store predecessor[v] when discovering v if a path must be reconstructed. FIFO ensures vertices at distance d are explored before later distance layers.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Addis Ababa University mock exam · Question 73

Question 14 | Q00214

Suppose we define a static method totalEvenLoc() that returns the total for even-index items of an integer list as shown, on the right, below. What is the Big O running time of the method if lst is a LinkedList of size N?

```
public static int totalEvenLoc(List<Integer> lst) {
int sum = 0;
for(int i = 0; i < lst.size(); i += 2)
sum += lst.get(i);
return sum;
}
```

A. O(N)

B. O(1)

C. O(N^2)

D. O(logN)

E. O(NlogN)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: O(N^2), tightly Θ(N^2)

The loop visits about N/2 even indices, but LinkedList.get(i) walks from the nearer endpoint. A linear number of those indices lie near the middle, each requiring linear traversal. DSA07 analyzes exactly this access pattern and obtains Θ(N^2) total work. Incrementing by two changes a constant factor, not the quadratic order.

Why each choice is right or wrong

A - Incorrect: O(N) would require constant-cost indexed access; LinkedList.get(i) instead traverses nodes.

B - Incorrect: A growing number of accesses and additions prevents a constant bound on total work.

C - Correct: A linear number of middle-index accesses each traverse a linear number of nodes, yielding quadratic time.

D - Incorrect: Neither the iteration count nor the accumulated linked-node traversal has logarithmic growth.

E - Incorrect: Quadratic traversal grows faster than N log N, so this is not a valid upper bound.

Primary note proof

Study-note section DSA07

> Indexed access is not constant for every List implementation. Java LinkedList.get(i) walks from the nearer endpoint. Summing get(0), get(2), get(4), ... over a length-N list still gives Theta(N^2), because a linear number of middle indices each require linear traversal. An iterator scans once in linear time. An exact executable runtime requires valid code and an explicit cost convention.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
> to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 26-34

III. Displaying the list of Nodes
After adding nodes to the list, we need to display the list of nodes on the screen. Here are the steps required to display all nodes item in the list:
1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the next pointer of the node it is currently indicating.
5. Jump back to step 2.

Data Structure and Algorithms module | PDF page 3 | lines 2-9

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:
II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n).
III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth.

Question sources

Mock exam 4 · Question 16

Question 15 | Q00215

The array 10, 8, 6, 2, 1, 4, 5 is organized into a full heap (heap and full BT). Which array represents the heap after the first and second biggest elements are removed?

A. 6 5 4 2 1

B. 6 4 5 2 1

C. 6 4 5 1 2

D. 6 5 4 1 2

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: 6 5 4 2 1

Remove 10 by moving 5 to root and adjusting: [8,5,6,2,1,4]. Remove 8 by moving 4 to root and adjusting through child 6: [6,5,4,2,1]. These are standard root-last swaps and downward heap restoration. Evidence: assigned PDF pages 11.

Why each choice is right or wrong

A - Correct: Matches both root deletions and adjustment steps. See assigned PDF 11.

B - Incorrect: A valid max heap, but not the result of these root-last adjustments. See assigned PDF 11.

C - Incorrect: Changes the lower pair 2,1 as well as giving a different adjustment history. See assigned PDF 11.

D - Incorrect: Reverses the unchanged lower pair 2,1. See assigned PDF 11.

Primary note proof

Study-note section DSA24

> Insert at the next complete-tree position and swap upward while parent order is violated. To remove the root, move the final element to root, reduce heap size, then swap downward with the more dominant child until order holds. Each path has logarithmic length. Building bottom-up by sinking internal nodes takes linear total time; repeated independent insertions give an O(n log n) construction bound.

Study-note section DSA24

> Max-heap [10,8,6,2,1,4,5]: remove 10, move 5 to root and sink, giving [8,5,6,2,1,4]. Remove 8, move 4 up then exchange with 6: [6,5,4,2,1].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

> iii. Heap Sort
> Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
> each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
> Algorithm:
> 1. Construct a binary tree
> • The root node corresponds to Data[0].
> • If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
> 2. Construct the heap tree from initial binary tree using "adjust" process.
> Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Mock exam 4 · Question 17

Question 16 | Q00216

Think of a reference implementation of a classical queue with front and rear pointing to the appropriate locations in the queue. Which of these references will change during an insertion into a possibly empty queue?

A. Neither change

B. Only front changes

C. None of the choices is correct

D. Both may change.

E. Only rear changes

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: Both may change.

Insertion into an empty linked queue makes the new node both front and rear. Insertion into a nonempty queue changes rear while front remains. Consequently both references may change across the allowed cases. Evidence: assigned PDF pages 7, 8, 5.

Why each choice is right or wrong

A - Incorrect: Insertion must update at least rear to include the new node. See assigned PDF 7, 8, 5.

B - Incorrect: Rear changes too; front alone is insufficient. See assigned PDF 7, 8, 5.

C - Incorrect: D describes the empty-queue case correctly. See assigned PDF 7, 8, 5.

D - Correct: Both change when the first node is inserted. See assigned PDF 7, 8, 5.

E - Incorrect: Rear alone changes only in the nonempty case. See assigned PDF 7, 8, 5.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs

if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 8 | lines 35-37

III. Linked list implementation of enqueue and dequeue operations
Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

Data Structure and Algorithms module | PDF page 5 | lines 13-24

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.
if (start == NULL) // If list is empty, then new node become start node
start = temp;
else {
temp2 = start; // list not empty! Thus temp2 starts from first node
while (temp2->next!= NULL) { // till the current last node found navigates forward
temp2 = temp2->next; // Move to next link in chain
}
temp2->next = temp; // new node (temp) appended at the end of the list

Question sources

Mock exam 4 · Question 18

Question 17 | Q00217

Binary search is to operate on an already ordered sequence of comparable keys through an indexed-access interface. Which listed property is required of its search domain, rather than being a particular physical array representation or allocation policy?

A. The array must hold elements of the same data type.

B. All are additional requirements.

C. The array must be contiguous.

D. The search interval must have valid finite bounds.

E. The array must fix its size a prior to population.

F. None is an additional requirement.

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: D. The search interval must have valid finite bounds.

Binary search maintains an interval of candidate indices and repeatedly shrinks it until a match or an empty interval is obtained. That operation requires valid finite bounds. Homogeneous physical storage and contiguity are properties of ordinary array representations, rather than requirements of the abstract interval-halving procedure. Fixing capacity before population is unnecessary: a resizable sequence can be populated first and searched afterward. Hence not all listed properties are requirements.

Why each choice is right or wrong

A - Incorrect: Comparable keys are assumed; identical physical data types are not the abstract domain requirement.

B - Incorrect: All fails because prior fixed allocation is unnecessary.

C - Incorrect: Contiguity is a representation choice rather than the abstract search-domain requirement.

D - Correct: The candidate index domain needs valid finite bounds.

E - Incorrect: A resizable sequence may be populated before searching.

F - Incorrect: Valid finite bounds are a required condition among these choices.

Primary note proof

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

Mock exam 4 · Question 19

Question 18 | Q00218

What will be the content of the queue (shown from queue front on the left to queue rear on the right), after the given sequence of operations shown below?

```
Queue q = new Queue();
q.enqueue(4);
q.enqueue(8);
if(q.peek() > 5)
q.dequeue();
q.enqueue(1);
q.enqueue(6);
q.dequeue();
```

A. 4, 6

B. 4, 8, 1

C. 8, 1, 6

D. 8, 6

E. 4, 1

Hint

Trace the initial state and each executed operation in order, preserving the requested output layout.

Answer C: 8, 1, 6

After enqueues the queue is [4,8]. Peek examines the front 4, so 4>5 is false and the conditional dequeue does not execute. Enqueues give [4,8,1,6]; the final dequeue removes 4, leaving [8,1,6]. Evidence: assigned PDF pages 7, 6.

Why each choice is right or wrong

A - Incorrect: Drops 8 and 1 without corresponding deletions. See assigned PDF 7, 6.

B - Incorrect: Leaves the removed 4 and omits the enqueued 6. See assigned PDF 7, 6.

C - Correct: Matches the final FIFO sequence. See assigned PDF 7, 6.

D - Incorrect: Incorrectly removes 1 as well as 4. See assigned PDF 7, 6.

E - Incorrect: Drops 8 and 6 and retains the removed 4. See assigned PDF 7, 6.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Mock exam 4 · Question 20

Question 19 | Q00286

What is the tightest listed worst-case time bound for constructing an optimal binary search tree using the standard interval dynamic program that tries every key as root of every interval?

A. O(n)

B. $O(n^3)$

C. O(logn)

D. O (n!)

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: B. $O(n^3)$

This asks about constructing an optimal BST, rather than searching a tree after construction. The standard dynamic program considers quadratically many key intervals. For each interval it tries up to a linear number of candidate roots, combining the relevant subproblem costs. Multiplying those counts gives cubic construction time. The cubic option states that bound; the linear, logarithmic and factorial alternatives do not give the tight construction order.

Why each choice is right or wrong

A - Incorrect: Linear time does not cover all intervals and root candidates.

B - Correct: Quadratic intervals times linear root trials give cubic time.

C - Incorrect: Logarithmic time concerns balanced search paths, not this construction.

D - Incorrect: Factorial is a loose upper bound rather than the tight listed order.

Primary note proof

Study-note section DSA19

> Standard optimal-BST dynamic programming tries each possible root for each key interval: $O(n^2)$ intervals times $O(n)$ roots gives $O(n^3)$ construction.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

> • Binary tree: a tree in which each node has at most two children called left child and right child.
> • Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
> satisfies the following.
> o Every node has a key and no two elements have the same key.
> o The keys in the right subtree are larger than the keys in the root.
> o The keys in the left subtree are smaller than the keys in the root.
> o The left and the right subtrees are also binary search trees.

Question sources

Computer Science model exam · Question 17

University of Gondar model exam · Question 17

Question 20 | Q00288

Which of the following is the best case time complexity of linear search algorithm?

A. O(n'2)

B. O(n)

C. O(1)

D. O(log n)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: O(1)

A match at the very first inspected element ends scanning after one comparison, independent of n. Interpret the options as tight growth orders rather than arbitrary loose Big-O upper bounds. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: Quadratic is not the tight best-case count. See assigned PDF 3.

B - Incorrect: Linear is the complete-scan bound, not the immediate-match cost. See assigned PDF 3.

C - Correct: One first-element comparison is constant. See assigned PDF 3.

D - Incorrect: Logarithmic describes halving, not immediate linear-search success. See assigned PDF 3.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first element
> • Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science model exam · Question 19

University of Gondar model exam · Question 19

Question 21 | Q00289

If the elements '1', '2', '3' and '4' are inserted in a queue, what would be order for the removal?

A. 4321

B. 3241

C. 1234

D. None of the above

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: 1234

FIFO removes the four insertions in their original order: 1 then 2 then 3 then 4. Evidence: assigned PDF pages 7.

Why each choice is right or wrong

A - Incorrect: 4321 reverses arrival order. See assigned PDF 7.

B - Incorrect: 3241 puts later arrivals before 1 and 2. See assigned PDF 7.

C - Correct: 1234 preserves insertion order. See assigned PDF 7.

D - Incorrect: C already supplies the FIFO result. See assigned PDF 7.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Computer Science model exam · Question 20

University of Gondar model exam · Question 20

Question 22 | Q00290

What is the disadvantage of array data structure?

A. The amount of memory to be allocated should be known beforehand.

B. Elements of an array can be accessed in constant time.

C. Elements are stored in contiguous memory blocks.

D. Multiple other data structures can be implemented using arrays.

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: The amount of memory to be allocated should be known beforehand.

The module calls array storage static and contrasts it with dynamically allocated nodes. A fixed array requires a capacity in advance; constant access and reuse as a storage implementation are benefits, while contiguous storage alone is not the capacity drawback. Evidence: assigned PDF pages 6, 7, 5.

Why each choice is right or wrong

A - Correct: Preallocated fixed capacity is the disadvantage illustrated by static array implementations. See assigned PDF 6, 7, 5.

B - Incorrect: Constant indexed access is an advantage, not a disadvantage. See assigned PDF 6, 7, 5.

C - Incorrect: Contiguity itself does not state the capacity limitation. See assigned PDF 6, 7, 5.

D - Incorrect: Supporting other structures is useful rather than a disadvantage. See assigned PDF 6, 7, 5.

Primary note proof

Study-note section DSA02

> A fixed array has a chosen capacity; the choice need not occur at compile time. A resizable array allocates a larger block and copies when needed. Inserting in the middle shifts later elements, so it costs linear time. Appending into spare capacity is constant; a resizing append can be linear, though repeated geometric growth gives amortized constant append cost.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

> I. Array implementation (static memory).
> Push(n): Adding item at top

```
top is the current top of stack and n is its
maximum size
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;
```

Data Structure and Algorithms module | PDF page 7 | lines 18-23

I. Simple Array implementation (static memory).
A simple array structure can be used to implement enqueue & dequeue operations. Thus item will
be added at the end of the array whereas dequeue operation removes items from the front. In simple
array implementation, to add item it checks whether there is room at the rear or not. Therefore, even though there are free released spaces at the front ot the array the queue implementation might
generate overflow message. This is the key drawback to use simple array to implement Queue.

Data Structure and Algorithms module | PDF page 5 | lines 3-8

While adding a node to a list, the chain needs to be rearranged properly so that the each nodes are connected to each other. It is possible to add a node anywhere in the chain (at the start, middle or end). Let's consider adding a node at the end of the list:
Step 1: First we need to reserve a space for the new node using the keyword new.
Node *temp = new *Node; // Reserving a space for temporary pointer which points to new node;

Question sources

Computer Science model exam · Question 21

University of Gondar model exam · Question 21

Question 23 | Q00291

For selection sort on n elements, n>=2, count only exchanges between distinct positions and omit the final one-element suffix. What is the maximum number of swaps?

A. n - 1

B. 1

C. n

D. n - 2

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. n - 1

Selection sort places the minimum remaining element into each successive position. After n-1 such placements, the last position is determined automatically. Each placement needs at most one exchange between distinct positions, so the maximum is n-1. This count excludes self-swaps and the redundant final-position call that some implementations include. The choice of input can require an exchange on every one of those meaningful passes.

Why each choice is right or wrong

A - Correct: There are n-1 meaningful placements, each requiring at most one exchange.

B - Incorrect: More than one exchange can be needed for larger n.

C - Incorrect: n counts an extra redundant final swap call under another convention.

D - Incorrect: n-2 omits a possible exchange on a meaningful pass.

Primary note proof

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i]. There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order. At most n-1 exchanges of distinct positions are needed. A loop that also swaps the last item with itself can count n swap calls.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 44-48

> B. Selection Sort
> Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n)

Question sources

Computer Science model exam · Question 22

University of Gondar model exam · Question 22

Question 24 | Q00293

In an infix-to-prefix conversion algorithm that temporarily holds pending operators and parentheses using last-in, first-out access, which abstract data type supplies that access discipline?

A. Stack

B. Queue

C. Binary tree

D. Linked list

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. Stack

Pending operators must be retrieved in reverse order when grouping or precedence requires them to be emitted. A stack provides that last-in, first-out discipline. An expression tree is another possible representation for conversion, but it is not the requested pending-operator ADT in this specified algorithm. A queue retrieves the oldest entry first, while a linked list alone describes storage links without prescribing the needed access policy.

Why each choice is right or wrong

A - Correct: A stack retrieves the latest pending operator first.

B - Incorrect: A queue uses first-in, first-out access.

C - Incorrect: An expression tree can support another conversion method but is not this pending-operator ADT.

D - Incorrect: A linked list can implement storage but alone does not specify LIFO behavior.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA13

To evaluate postfix, push operands. At a binary operator, pop right operand first, then left, and push left op right. Build an expression tree for either prefix (preorder) or postfix (postorder); a stack is useful but is not the only possible conversion representation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 46-52

Applications of Stack
• It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
• Infix to Postfix Transformation
• It is useful during the execution of recursive programs
• A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
• A stack (memory stack) can be used in function calls including recursion.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Question sources

Computer Science model exam · Question 24

University of Gondar model exam · Question 24

Question 25 | Q00294

Which one of the following is the process of inserting an element in the stack?

A. Insert

B. Add

C. Push

D. None of the above

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Push

The stack terminology explicitly names adding an element Push. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: Insert is a generic action rather than the given operation name. See assigned PDF 6.

B - Incorrect: Add is descriptive rather than the module stack name. See assigned PDF 6.

C - Correct: Push is defined as adding an element to the stack. See assigned PDF 6.

D - Incorrect: C is explicitly defined, so none is false. See assigned PDF 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science model exam · Question 25

University of Gondar model exam · Question 25

Question 26 | Q00371

How many cases are there, which are used to compare various data structure's execution time in a relative manner?

A. 2

B. 3

C. 4

D. 5

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: Three: best, average, worst.

DSA05 names best, average, and worst as the ordinary cases for comparing costs on inputs of equal size. Average additionally needs an input distribution. Counting these cases gives three. This is a classification of input cases, rather than a count of resources, data structures, or asymptotic bound notations.

Why each choice is right or wrong

A - Incorrect: Two omits one of the three named categories: best, average, and worst case.

B - Correct: The note lists three ordinary fixed-size input cases: best, average, and worst.

C - Incorrect: Four does not match the three ordinary input-case categories listed in DSA05.

D - Incorrect: Five adds categories outside the stated best, average, and worst input-case classification.

Primary note proof

Study-note section DSA-S01

> Fix the input size before comparing cases. Best-case cost is the minimum resource use over legal inputs of that size; worst-case cost is the maximum. Average-case cost averages costs using a stated input distribution. These three input-case functions can each be described using upper, lower or tight asymptotic bounds; the case name does not select the bound notation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 33-37

> Algorithm analysis refers to the process of determining the amount of computing time and storage
> space required by different algorithms. It used to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexityof the algorithm.
> Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

Question sources

Data Structures & Algorithms practice questions · Question 1

Question 27 | Q00372

Which case of data structure operation takes maximum time?

A. Worst Case

B. Average Case

C. Best Case

D. None of the above

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer A: Worst case

DSA05 distinguishes best, average, and worst over inputs of the same size. Worst selects the largest runtime; best selects the smallest, while average takes an expectation under a distribution. Therefore Worst Case directly matches maximum time. This input-case distinction is independent of whether the resulting cost function is described with O, Ω, or Θ notation.

Why each choice is right or wrong

A - Correct: Worst case takes the maximum operation cost over valid inputs of the same size.

B - Incorrect: Average case computes an expected cost under a distribution instead of selecting the maximum.

C - Incorrect: Best case selects the minimum cost, the opposite extremum from the question.

D - Incorrect: Worst Case is offered and directly identifies the requested maximum-time input case.

Primary note proof

Study-note section DSA-S01

> Fix the input size before comparing cases. Best-case cost is the minimum resource use over legal inputs of that size; worst-case cost is the maximum. Average-case cost averages costs using a stated input distribution. These three input-case functions can each be described using upper, lower or tight asymptotic bounds; the case name does not select the bound notation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 33-37

> Algorithm analysis refers to the process of determining the amount of computing time and storage
> space required by different algorithms. It used to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexityof the algorithm.
> Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

Question sources

Data Structures & Algorithms practice questions · Question 2

Question 28 | Q00375

Which listed characteristic is a binary-tree shape condition rather than a general functional-correctness or resource-cost criterion for arbitrary data-structure operations? Interpret completeness as the complete-tree shape property.

A. Completeness

B. Correctness

C. Time Complexity

D. Space Complexity

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. Completeness

Correctness asks whether operations produce their specified results. Time and space complexity measure operation counts and memory use under a resource model. Completeness, under the explicit meaning supplied here, instead describes how a binary tree's levels are filled: earlier levels are full and the last fills from the left. It is a particular shape condition, rather than a universal assessment criterion for every data structure.

Why each choice is right or wrong

A - Correct: Completeness describes a particular binary-tree shape.

B - Incorrect: Correctness assesses whether an operation meets its specification.

C - Incorrect: Time complexity assesses operation cost.

D - Incorrect: Space complexity assesses memory cost.

Primary note proof

Study-note section DSA18

> Binary means at most two children. Full means each node has zero or two children. Complete means all earlier levels are filled and the last fills from left. Perfect means all leaves share a level and every internal node has two children. Balanced means height stays logarithmic under a stated rule; it does not require a perfect shape.

Study-note section DSA04

> Finiteness means termination after finitely many steps for every valid input. Definiteness means each instruction has one interpretation; effectiveness means each step is executable. Correctness means meeting the stated result for every valid input. A finite program can still return the wrong answer.

Study-note section DSA05

> Time counts chosen operations; space counts memory, including live recursive frames.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 33-37

> Algorithm analysis refers to the process of determining the amount of computing time and storage
> space required by different algorithms. It used to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexityof the algorithm.
> Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

Question sources

Data Structures & Algorithms practice questions · Question 5

Question 29 | Q00377

Which abstract data types permit deleting an element at the front and inserting an element at the rear? Include types that also permit additional endpoint operations. Select all that apply.

A. Stacks

B. Queues

C. Deques

D. Binary search tree

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B, C: B. Queues; C. Deques

An ordinary queue supplies exactly the stated FIFO operations: insertion at the rear and removal at the front. A deque allows insertion and removal at both ends, which includes those same two operations as a subset. The question explicitly includes types with additional endpoint capabilities, so both qualify. A stack exposes a single LIFO top, and a BST organizes access by keys rather than front and rear.

Why each choice is right or wrong

A - Incorrect: A stack exposes a single LIFO top rather than these FIFO endpoints.

B - Correct: A queue enqueues at rear and dequeues at front.

C - Correct: A deque includes both requested operations among its endpoint capabilities.

D - Incorrect: A BST uses key-based tree operations rather than front/rear endpoints.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Study-note section DSA16

> A deque permits both-end insertion/removal without guaranteeing middle access. Endpoint-linked operations are constant but middle indexing is linear. A circular-array deque may support indexed access. Dequeue names removal; deque names the structure.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

> Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
> Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Question sources

Data Structures & Algorithms practice questions · Question 7

Question 30 | Q00379

Which algorithms have tight worst-case time Theta(n^2)? Use ordinary array comparison-sort implementations; radix sort uses counting-based passes with fixed digit count and fixed radix. Select all that apply.

A. Merge sort

B. Insertion sort

C. Quick sort

D. Radix sort

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer B, C: B. Insertion sort; C. Quick sort

Insertion sort on reverse input shifts a growing sorted prefix, totaling quadratic work. Quicksort repeatedly choosing an extreme pivot also accumulates quadratic partition work. Ordinary merge sort instead performs linear work at logarithmically many levels. Under the explicitly fixed digit count and radix, radix sorting is linear in n. Tight bounds distinguish these actual growth rates from the looser quadratic upper bounds that faster algorithms also satisfy.

Why each choice is right or wrong

A - Incorrect: Ordinary merge sort has tight n log n worst time.

B - Correct: Reverse-order insertion gives quadratic shifts.

C - Correct: Repeated extreme quicksort pivots give quadratic work.

D - Incorrect: Fixed-width, fixed-radix counting passes give linear time.

Primary note proof

Study-note section DSA34

> Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Study-note section DSA36

> Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work.

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA39

> With d digits and radix r, counting-based passes take O(d(n+r)) time and O(n+r) auxiliary storage.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 34-43

> A. Insertion Sort
> The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
> them in order. Then add 3rd
> item to the relatively sorted portion and find its correct location
> (relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
> in the list.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n2
>)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 9

Question 31 | Q00380

In which sorting algorithm does the largest element "bubble" to the end of the list at each pass?

A. Bubble sort

B. Selection sort

C. Insertion sort

D. Heap sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Bubble sort

Repeated adjacent out-of-order swaps push the largest item toward the final position in a left-to-right pass. This is exactly the bubble-sort mechanism. Evidence: assigned PDF pages 4, 3, 11.

Why each choice is right or wrong

A - Correct: Adjacent comparisons bubble the largest to its proper end. See assigned PDF 4, 3, 3, 11.

B - Incorrect: Selection finds a minimum and swaps it into position rather than bubbling through adjacent pairs. See assigned PDF 4, 3, 3, 11.

C - Incorrect: Insertion inserts each new element into a sorted prefix. See assigned PDF 4, 3, 3, 11.

D - Incorrect: Heap sort removes a root maximum using tree adjustment. See assigned PDF 4, 3, 3, 11.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right. The largest remaining value moves to the final unsorted position after one left-to-right pass. Shrink the next pass by one position. Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i].

Study-note section DSA34

> For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

> C. Bubble Sort
> The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
> the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n2
>)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

> B. Selection Sort
> Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

> A. Insertion Sort
> The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
> them in order. Then add 3rd

item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Data Structures & Algorithms practice questions · Question 10

Question 32 | Q00381

Which algorithms have tight best-case time Theta(n)? Use early-exit bubble sort, ordinary quicksort, ordinary array mergesort without merge-skipping, and standard insertion sort. Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer A, D: A. Bubble sort; D. Insertion sort

Early-exit bubble sort terminates after one complete no-swap pass on already sorted input, so its best case is linear. Insertion sort also checks each new item once without shifting it on sorted input. Ordinary quicksort still partitions across logarithmically many balanced levels in its best case, and ordinary mergesort still performs its merges. Thus exactly the bubble and insertion implementations have the requested linear best case.

Why each choice is right or wrong

A - Correct: The no-swap early exit gives a linear sorted-input pass.

B - Incorrect: Ordinary quicksort has n log n best growth.

C - Incorrect: Without merge-skipping, mergesort still uses n log n work.

D - Correct: Sorted input needs one ordering check per insertion pass.

Primary note proof

Study-note section DSA32

> Early stopping makes a sorted input linear: one no-swap pass finishes.

Study-note section DSA34

> On sorted input each pass needs one ordering check and no shifts: linear time.

Study-note section DSA36

> Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work.

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Related reading (context, not primary note proof)

Study-note section DSA33

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

> C. Bubble Sort
> The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
> the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n2
>)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Variant 1 | Alternative version 1

Which algorithms have tight best-case time Theta(n)? Use early-exit bubble sort, standard selection sort, ordinary array mergesort without merge-skipping, and standard insertion sort. Select all that apply.

A. Bubble sort

B. Selection sort

C. Merge sort

D. Insertion sort

Answer A, D: A. Bubble sort; D. Insertion sort

Early-exit bubble sort needs one no-swap pass on sorted input, while insertion sort performs one successful ordering check per new item with no shifts. Both have linear best time. Selection sort still scans every remaining suffix and makes quadratic comparisons, even on sorted input. Ordinary mergesort retains logarithmically many levels of merging. Consequently A and D are the complete selection for this variant.

Why each choice is right or wrong

A - Correct: An early-exit no-swap pass makes optimized bubble sort linear on sorted input.

B - Incorrect: Selection sort still scans each suffix and makes quadratic comparisons even when sorted.

C - Incorrect: Ordinary merge sorting does n log n work across input orders.

D - Correct: Insertion sort is linear on sorted input because each pass needs no shifts.

Primary note proof

Study-note section DSA32

> Early stopping makes a sorted input linear: one no-swap pass finishes.

Study-note section DSA34

> On sorted input each pass needs one ordering check and no shifts: linear time.

Study-note section DSA33

> There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order.

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Related reading (context, not primary note proof)

Study-note section DSA36

Question sources

Data Structures & Algorithms practice questions · Question 11

Computer Science MCQ practice collection · Question 454

Question 33 | Q00382

Which ordinary iterative array sorting implementations use constant auxiliary element storage? Use buffered mergesort for C. Select all that apply.

A. Bubble sort

B. Selection sort

C. Merge sort

D. Insertion sort

Hint

Apply the naming, type and syntax rules before deciding whether an option can be used as stated.

Answer A, B, D: A. Bubble sort; B. Selection sort; D. Insertion sort

Bubble sort reuses one temporary element for each adjacent swap. Selection sort retains the minimum position while scanning, then exchanges elements using the same fixed-size temporary storage. Insertion sort saves one item and shifts preceding entries within the original array before filling the gap. The indices and saved elements stay fixed in number as n grows, giving constant auxiliary storage for A, B and D. Buffered mergesort instead needs a linear auxiliary buffer.

Why each choice is right or wrong

A - Correct: Adjacent swaps reuse a single temporary element and a fixed number of indices.

B - Correct: The suffix scan retains a minimum index; the exchange needs only fixed-size temporary storage.

C - Incorrect: The specified buffered array mergesort allocates a linear auxiliary merge buffer.

D - Correct: One saved item and fixed-size indices suffice while larger entries shift within the original array.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right.

Study-note section DSA34

> For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i].

Study-note section DSA37

> Ordinary array merging needs a linear auxiliary buffer.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 12

Question 34 | Q00383

Which sorting algorithm uses a pivot element to partition the list into two sublists?

A. Bubble sort

B. Selection sort

C. Quick sort

D. Insertion sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Quick sort

The source quicksort procedure chooses a pivot and partitions around it before recursively sorting both sides. Evidence: assigned PDF pages 11, 4, 3.

Why each choice is right or wrong

A - Incorrect: Bubble uses adjacent exchanges without a pivot partition. See assigned PDF 11, 4, 3, 3.

B - Incorrect: Selection chooses the minimum for its next position. See assigned PDF 11, 4, 3, 3.

C - Correct: Quicksort explicitly chooses and positions a pivot. See assigned PDF 11, 4, 3, 3.

D - Incorrect: Insertion extends a sorted prefix rather than partitioning by a pivot. See assigned PDF 11, 4, 3, 3.

Primary note proof

Study-note section DSA36

> Choose a pivot, partition, then recursively sort the pieces. Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work. Expected n log n needs an input/random-pivot model. Random and median-of-three choices have different tradeoffs; neither is universally best.

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right.

Study-note section DSA34

> Begin with the first element already sorted.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

> ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).

Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Data Structures & Algorithms practice questions · Question 13

Question 35 | Q00384

Which sorting implementations preserve the relative input order of equal keys? Use bubble sort with strict adjacent swaps, standard exchange-based selection sort, ordinary in-place quicksort, and mergesort with left-first tie handling. Select all that apply.

A. Bubble sort

B. Selection sort

C. Quick sort

D. Merge sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A, D: A. Bubble sort; D. Merge sort

Strict bubble swaps never exchange equal adjacent keys, so equal records retain their input order. Left-first handling during merging likewise preserves the order of ties from the left and right runs. Standard selection sort can reverse ties through a distant minimum exchange, and ordinary quicksort partitioning does not guarantee stable order. The specified implementation details therefore make bubble and merge the two stable choices.

Why each choice is right or wrong

A - Correct: Strict adjacent swaps preserve equal-key order.

B - Incorrect: A distant minimum exchange can reverse equal records.

C - Incorrect: Ordinary in-place partitioning does not guarantee stability.

D - Correct: Taking the left run first on ties preserves equal-key order.

Primary note proof

Study-note section DSA32

> Strict swaps preserve equal-key order.

Study-note section DSA37

> With left-first tie handling, equal keys retain order.

Study-note section DSA33

> Selection suits expensive record moves with cheap key comparisons, not every large dataset. It is unstable: exchanging first 2a with 1 in [2a,2b,1] gives [1,2b,2a]. Constant extra element storage is shared by several sorts.

Study-note section DSA31

> stack usually log n average, n worst; unstable

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in
the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 14

Question 36 | Q00385

Which algorithm has tight Theta(n log n) time in both the average and worst cases? Use ordinary array implementations; radix sort uses fixed digit count and fixed radix.

A. Bubble sort

B. Selection sort

C. Merge sort

D. Radix sort

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: C. Merge sort

Ordinary mergesort performs linear merging at each of logarithmically many levels, giving n log n work regardless of input order. Bubble and selection have quadratic average and worst time. Fixed-width, fixed-radix counting passes instead give radix sort linear time, so radix does not have the requested tight n log n growth. Specifying tight bounds excludes faster methods that would also satisfy the looser Big-O upper bound.

Why each choice is right or wrong

A - Incorrect: Bubble has quadratic average and worst time.

B - Incorrect: Selection makes quadratic comparisons in every order.

C - Correct: Ordinary mergesort has tight n log n average and worst time.

D - Incorrect: With fixed digits and radix, radix sorting has tight linear time.

Primary note proof

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA39

> With d digits and radix r, counting-based passes take O(d(n+r)) time and O(n+r) auxiliary storage.

Study-note section DSA32

> Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Study-note section DSA33

> There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order.

Related reading (context, not primary note proof)

Study-note section DSA34

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Variant 1 | Alternative version 1

Under ordinary array comparison-sort implementations, which algorithm has tight Theta(n log n) time in every input case? Exclude optimizations that skip work on already ordered input.

A. Bubble sort

B. Selection sort

C. Merge sort

D. Insertion sort

Answer C: C. Merge sort

Buffered mergesort has linear merge work at logarithmically many levels in every input case, giving tight n log n average and worst growth. Bubble, selection and insertion have quadratic worst time, and their usual average time is also quadratic. Quicksort's extreme partitions can violate the requested worst-case bound, whereas buffered mergesort meets it.

Why each choice is right or wrong

A - Incorrect: Bubble has quadratic average and worst time.

B - Incorrect: Selection always makes quadratic comparisons.

C - Correct: Mergesort has the required tight average and worst bounds.

D - Incorrect: Insertion has quadratic average and worst time.

Primary note proof

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA32

> Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Study-note section DSA33

> There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order.

Study-note section DSA34

> Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Related reading (context, not primary note proof)

Study-note section DSA39

Question sources

Data Structures & Algorithms practice questions · Question 15

Computer Science MCQ practice collection · Question 438

Question 37 | Q00386

For a large array of nonnegative integer keys with fixed digit count and fixed radix, which listed ordinary algorithm can achieve linear time using counting-based digit passes and O(n+r) auxiliary storage?

A. Bubble sort

B. Selection sort

C. Radix sort

D. Insertion sort

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: C. Radix sort

With digit count d and radix r fixed, the radix cost O(d(n+r)) reduces to linear growth in n. Its counting-based stable passes use auxiliary storage for output and digit counts. The key assumptions make this an eligible radix-sorting workload rather than an unspecified large-list claim. Bubble, selection and insertion use constant extra element storage in their ordinary forms and can require quadratic work.

Why each choice is right or wrong

A - Incorrect: Bubble uses adjacent comparisons and can require quadratic work.

B - Incorrect: Selection uses quadratic comparisons and constant extra element storage.

C - Correct: Radix uses the stated digit passes and auxiliary storage.

D - Incorrect: Insertion uses prefix shifts and can require quadratic work.

Primary note proof

Study-note section DSA39

> With d digits and radix r, counting-based passes take O(d(n+r)) time and O(n+r) auxiliary storage.

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right.

Study-note section DSA34

> For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Data Structures & Algorithms practice questions · Question 16

Question 38 | Q00387

Which listed sorting algorithm orders fixed-length strings by stably processing individual character positions, without using whole-string comparison operations? Assume an ordered finite character alphabet and LSD passes.

A. Bubble sort

B. Selection sort

C. Merge sort

D. Radix sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: D. Radix sort

Strings can be compared lexicographically, but the stated question asks for a digit-based mechanism rather than whole-key comparisons. LSD radix sorting treats each character position as a digit, stably sorting from the last position toward the first. Stability retains the ordering established by earlier passes. Bubble, selection and merge use comparisons between keys in their ordinary forms, so they do not implement this specified character-pass mechanism.

Why each choice is right or wrong

A - Incorrect: Ordinary bubble sort compares adjacent whole keys.

B - Incorrect: Ordinary selection chooses a minimum through key comparisons.

C - Incorrect: Ordinary merge compares the fronts of sorted key runs.

D - Correct: LSD radix uses stable character-position passes from least significant to most significant.

Primary note proof

Study-note section DSA39

> LSD radix sorting handles fixed-width digit keys by stably sorting least-significant digit first, then moving left. Stable intermediate passes preserve the ordering already established by less-significant digits. With d digits and radix r, counting-based passes take O(d(n+r)) time and O(n+r) auxiliary storage.

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right.

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i].

Study-note section DSA37

> Split into halves, sort them, then merge by advancing the smaller current front.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

> C. Bubble Sort
> The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
> the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
> • How many comparisons? => O(n2

) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 17

Question 39 | Q00388

How many passes does an insertion sort algorithm consist of?

A. N

B. N-1

C. N+1

D. N2

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: N-1

The first element is already a sorted prefix. Each of the remaining N-1 elements is inserted once: the source starts with the first two, then adds the third, and so on. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: N counts the initial singleton as an insertion pass. See assigned PDF 3.

B - Correct: There is one insertion for each position 2 through N. See assigned PDF 3.

C - Incorrect: N+1 adds two passes with no corresponding elements. See assigned PDF 3.

D - Incorrect: N squared is a work bound, not the count of outer passes. See assigned PDF 3.

Primary note proof

Study-note section DSA34

> Begin with the first element already sorted. For each remaining item, save its value, shift larger preceding values right, and insert it into the gap. That gives n-1 insertion passes. On sorted input each pass needs one ordering check and no shifts: linear time. Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 34-43

> A. Insertion Sort
> The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
> them in order. Then add 3rd
> item to the relatively sorted portion and find its correct location
> (relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
> in the list.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n2
>)

Question sources

Data Structures & Algorithms practice questions · Question 18

Question 40 | Q00389

What is the running time of an insertion sort algorithm if the input is pre-sorted?

A. O(N2)

B. O(N log N)

C. O(N)

D. O(M log N)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: O(N)

For sorted input every new item already belongs after its predecessor. Each of N-1 prefix extensions needs constant comparison work and no shifts, giving Theta(N). Big-O choices are interpreted as tight orders. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: Quadratic is the adverse-input bound, not the tight sorted-input cost. See assigned PDF 3.

B - Incorrect: N log N adds work not performed by this insertion process. See assigned PDF 3.

C - Correct: One constant check per extension gives linear work. See assigned PDF 3.

D - Incorrect: M is undefined and no M-dependent process is given. See assigned PDF 3.

Primary note proof

Study-note section DSA34

> Begin with the first element already sorted. For each remaining item, save its value, shift larger preceding values right, and insert it into the gap. That gives n-1 insertion passes. On sorted input each pass needs one ordering check and no shifts: linear time. Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 34-43

> A. Insertion Sort
> The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
> them in order. Then add 3rd
> item to the relatively sorted portion and find its correct location
> (relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
> in the list.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n2
>)

Question sources

Data Structures & Algorithms practice questions · Question 19

Question 41 | Q00390

Which of the following examples represent the worst case input for an insertion sort?

A. array in sorted order

B. array sorted in reverse order

C. normal unsorted array

D. large array

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: array sorted in reverse order

Reverse order forces each inserted item to move through the whole preceding prefix. The movement total 1+2+...+(n-1)=n(n-1)/2 is quadratic. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: Sorted input needs no shifts. See assigned PDF 3.

B - Correct: Each next element is smaller than every element in the prefix. See assigned PDF 3.

C - Incorrect: An arbitrary unsorted array need not maximize all insertion distances. See assigned PDF 3.

D - Incorrect: Large size alone does not determine relative ordering or maximal shifts. See assigned PDF 3.

Primary note proof

Study-note section DSA34

> Begin with the first element already sorted. For each remaining item, save its value, shift larger preceding values right, and insert it into the gap. That gives n-1 insertion passes. On sorted input each pass needs one ordering check and no shifts: linear time. Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 34-43

> A. Insertion Sort
> The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
> them in order. Then add 3rd
> item to the relatively sorted portion and find its correct location
> (relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item

in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Data Structures & Algorithms practice questions · Question 20

Question 42 | Q00391

Which scenario specifically favors selection sort's small number of record exchanges, when records themselves are moved and comparisons are cheap? Interpret large values as large records and small keys as inexpensive comparison keys.

A. The input is already sorted

B. A large file has to be sorted

C. Large values need to be sorted with small keys

D. Small values need to be sorted with large keys

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: C. Large values need to be sorted with small keys

Selection sort scans suffixes and makes quadratic comparisons regardless of the initial order, but it performs at most one meaningful exchange per placement. That tradeoff can be useful when physically moving a large record is expensive while comparing its small key is cheap. A large file or sorted input alone does not offer this advantage. Expensive large-key comparisons instead make selection's many comparisons less attractive.

Why each choice is right or wrong

A - Incorrect: Sorted order does not reduce selection's quadratic comparison count.

B - Incorrect: Large file size alone does not identify the expensive-move advantage.

C - Correct: Large records with cheap keys match the many-comparisons/few-exchanges tradeoff.

D - Incorrect: Expensive comparisons with cheap moves do not favor that tradeoff.

Primary note proof

Study-note section DSA33

At position i, scan the remaining suffix for its minimum, then exchange it with A[i]. There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order. At most n-1 exchanges of distinct positions are needed. A loop that also swaps the last item with itself can count n swap calls.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Data Structures & Algorithms practice questions · Question 21

Question 43 | Q00392

What is the advantage of selection sort over other sorting techniques?

A. It requires no additional storage space

B. It is scalable

C. It works best for inputs which are already sorted

D. It is faster than any other sorting technique

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Constant extra element storage, conventionally called in-place sorting.

DSA33 says selection uses constant extra element storage, which is the conventional meaning of A's no-additional-storage phrasing. It still needs a few temporary values, so literally zero bytes would be inaccurate. Its quadratic comparisons do not improve on already sorted input, and the note gives no universal scalability or fastest-algorithm claim. A names the valid conventional benefit.

Why each choice is right or wrong

A - Correct: Interpreted conventionally as in-place, selection uses constant extra element storage rather than an auxiliary array.

B - Incorrect: Quadratic comparisons do not provide an unqualified scalability advantage for growing datasets.

C - Incorrect: Selection makes the same quadratic comparisons even on already sorted input.

D - Incorrect: No universal fastest claim follows; selection trades few moves for many comparisons.

Primary note proof

Study-note section DSA33

> Selection suits expensive record moves with cheap key comparisons, not every large dataset. It is unstable: exchanging first 2a with 1 in [2a,2b,1] gives [1,2b,2a]. Constant extra element storage is shared by several sorts.

Study-note section DSA33

> There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 44-48

> B. Selection Sort
> Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

> A. Insertion Sort
> The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
> them in order. Then add 3rd
> item to the relatively sorted portion and find its correct location
> (relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item

in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 22

Question 44 | Q00393

What is an external sorting algorithm?

A. Algorithm that uses tape or disk during the sort

B. Algorithm that uses main memory during the sort

C. Algorithm that involves swapping

D. Algorithm that are considered 'in place'

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Sorting using external storage such as disk or tape.

DSA37 defines external sorting through datasets beyond main memory and buffered disk runs. Option A captures this distinguishing external-storage use. Main memory may still hold buffers, but using it alone does not define external sorting. Swapping and being in place are separate implementation properties and do not identify whether the working dataset resides externally.

Why each choice is right or wrong

A - Correct: Tape or disk stores external runs when the working dataset does not fit wholly in main memory.

B - Incorrect: Main-memory use alone does not distinguish external sorting; external algorithms also use memory for buffers.

C - Incorrect: Swapping is an operation used by many sorts and does not define external storage use.

D - Incorrect: In-place concerns auxiliary element storage, independently of whether sorting accesses an external dataset.

Primary note proof

Study-note section DSA37

> Top-down mergesort recurses; bottom-up merges runs of widths 1,2,4,... iteratively. In-place merging variants exist with different tradeoffs; linear merge names an operation, not a universally standardized sort variant. Internal sorting fits the working dataset in memory. External sorting builds sorted runs and merges buffered streams from disk, aiming to reduce I/O. Reverse order alone does not characterize maximum merge comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 33-37

> Algorithm analysis refers to the process of determining the amount of computing time and storage
> space required by different algorithms. It used to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexityof the algorithm.
> Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

Question sources

Data Structures & Algorithms practice questions · Question 23

Question 45 | Q00395

What is the worst case complexity of bubble sort?

A. O(nlogn)

B. O(logn)

C. O(n)

D. O(n2)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer D: O(n2)

An adversarial input requires repeated passes over adjacent pairs; the total comparisons and swaps are quadratic as stated in the module. Evidence: assigned PDF pages 4.

Why each choice is right or wrong

A - Incorrect: n log n understates the tight adverse-input work. See assigned PDF 4.

B - Incorrect: Logarithmic growth cannot account for repeated full passes. See assigned PDF 4.

C - Incorrect: Linear describes at most a single pass. See assigned PDF 4.

D - Correct: The source explicitly gives O(n2) comparisons and swaps. See assigned PDF 4.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right. The largest remaining value moves to the final unsorted position after one left-to-right pass. Shrink the next pass by one position. Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

> C. Bubble Sort
> The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
> the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
> • How many comparisons? => O(n2
>) and How many swaps? => O(n2
>)

Question sources

Data Structures & Algorithms practice questions · Question 25

Question 46 | Q00396

What is the average case complexity of bubble sort?

A. O(nlogn)

B. O(logn)

C. O(n)

D. O(n2)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer D: O(n^2) under the usual average-case model.

DSA32 identifies quadratic average and worst work from repeated adjacent comparisons. Under the usual random-permutation average model, bubble sort therefore takes $\Theta(n^2)$, which implies $O(n^2)$. Early stopping makes an already sorted input linear, but that is the optimized best case rather than the ordinary average over input arrangements.

Why each choice is right or wrong

A - Incorrect: Quadratic average work grows faster than n log n, so this bound is too small.

B - Incorrect: Repeated adjacent passes do not have logarithmic average growth under the usual input model.

C - Incorrect: Linear work is the optimized sorted-input best case, not ordinary average-case bubble sorting.

D - Correct: Quadratic average work is bounded by $O(n^2)$, matching the adjacent-pass analysis in DSA32.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right. The largest remaining value moves to the final unsorted position after one left-to-right pass. Shrink the next pass by one position. Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Study-note section DSA-S01

> Fix the input size before comparing cases. Best-case cost is the minimum resource use over legal inputs of that size; worst-case cost is the maximum. Average-case cost averages costs using a stated input distribution. These three input-case functions can each be described using upper, lower or tight asymptotic bounds; the case name does not select the bound notation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Data Structures & Algorithms practice questions · Question 26

Question 47 | Q00397

What is the auxiliary space complexity of merge sort?

A. O(1)

B. O(log n)

C. O(n)

D. O(n log n)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: O(n)

Merging both halves needs an auxiliary array holding up to n elements. Recursion contributes O(log n) frames, dominated by O(n) temporary element storage in the ordinary implementation. Evidence: assigned PDF pages 11.

Why each choice is right or wrong

A - Incorrect: Constant space contradicts the full merge array. See assigned PDF 11.

B - Incorrect: Logarithmic describes recursion depth alone. See assigned PDF 11.

C - Correct: A merge buffer proportional to n dominates auxiliary storage. See assigned PDF 11.

D - Incorrect: n log n counts cumulative work across levels, not simultaneously live storage when buffers are reused. See assigned PDF 11.

Primary note proof

Study-note section DSA37

Ordinary array merging needs a linear auxiliary buffer.

Study-note section DSA37

There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA07

An n-level ordinary recursion with constant live storage per level needs Theta(n) stack space.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 27

Question 48 | Q00398

What is the tightest listed worst-case time bound for ordinary array mergesort that performs all merges and does not skip already ordered runs? Interpret logarithms explicitly as parenthesized in the options.

A. O(n log n)

B. $O(n^2)$

C. $O(n^2 \log n)$

D. $O(n (\log n)^2)$

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. O(n log n)

Ordinary mergesort splits the array into logarithmically many levels and performs linear total merging work per level. It therefore has n log n worst-case growth under the specified implementation. Sorted input does not skip any merges here. The quadratic and $n^2 \log n$ choices are looser upper bounds, while $n(\log n)^2$ grows faster than n log n. Asking for the tightest listed bound selects A uniquely.

Why each choice is right or wrong

A - Correct: This matches the actual n log n growth.

B - Incorrect: Quadratic is a valid but looser upper bound.

C - Incorrect: $n^2 \log n$ is a valid but looser upper bound.

D - Incorrect: $n(\log n)^2$ grows faster and is a looser upper bound.

Primary note proof

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA06

> For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

> iv. Merge sort
> Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
> Algorithm:
> 1. Divide the array in to two halves.
> 2. Recursively sort the first n/2 items.
> 3. Recursively sort the last n/2 items.
> 4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 3 | lines 2-9

> I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:
> II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n).
> III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth.

Question sources

Data Structures & Algorithms practice questions · Question 28

Question 49 | Q00399

What is the tightest listed best-case time bound for ordinary array mergesort that performs all merges and does not skip already ordered runs? Interpret logarithms explicitly as parenthesized in the options.

A. O(n log n)

B. $O(n^2)$

C. $O(n^2 \log n)$

D. $O(n (\log n)^2)$

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. O(n log n)

Ordinary mergesort splits the array into logarithmically many levels and performs linear total merging work per level. It therefore has n log n best-case growth under the specified implementation. Sorted input does not skip any merges here. The quadratic and $n^2 \log n$ choices are looser upper bounds, while $n(\log n)^2$ grows faster than n log n. Asking for the tightest listed bound selects A uniquely.

Why each choice is right or wrong

A - Correct: This matches the actual n log n growth.

B - Incorrect: Quadratic is a valid but looser upper bound.

C - Incorrect: $n^2 \log n$ is a valid but looser upper bound.

D - Incorrect: $n(\log n)^2$ grows faster and is a looser upper bound.

Primary note proof

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA06

> For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 29

Question 50 | Q00400

Using the terminology of the course note, which listed phrase names the combining operation rather than a recognized organization or storage variant of mergesort?

A. in-place merge sort

B. bottom up merge sort

C. top down merge sort

D. linear merge sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: D. linear merge sort

Top-down mergesort recursively divides the sequence, while bottom-up mergesort iteratively combines runs of increasing widths. In-place variants change the storage tradeoff used while merging. Under the note's stated terminology, linear merge names the combining operation itself rather than a separate standardized sort variant. This is a distinction within the supplied naming convention, not a claim that no author could ever use that phrase differently.

Why each choice is right or wrong

A - Incorrect: In-place merging is a recognized storage variant.

B - Incorrect: Bottom-up mergesort is the iterative run-merging organization.

C - Incorrect: Top-down mergesort is the recursive organization.

D - Correct: Linear merge names the combining operation under the stated terminology.

Primary note proof

Study-note section DSA37

Top-down mergesort recurses; bottom-up merges runs of widths 1,2,4,... iteratively. In-place merging variants exist with different tradeoffs; linear merge names an operation, not a universally standardized sort variant. Internal sorting fits the working dataset in memory.

External sorting builds sorted runs and merges buffered streams from disk, aiming to reduce I/O. Reverse order alone does not characterize maximum merge comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 30

Question 51 | Q00401

Which of the following is not in place sorting algorithm by default?

A. merge sort

B. quick sort

C. heap sort

D. insertion sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: merge sort

The default merge algorithm explicitly uses an auxiliary array. The described partition, heap swaps and insertion can use the original array; recursion-stack use is distinct from requiring a second element array. Evidence: assigned PDF pages 11, 3.

Why each choice is right or wrong

A - Correct: Merge expressly allocates auxiliary array storage. See assigned PDF 11, 11, 11, 3.

B - Incorrect: Quicksort partitions in the existing array in its conventional form. See assigned PDF 11, 11, 11, 3.

C - Incorrect: Heap sorts by exchanging root and last within the array. See assigned PDF 11, 11, 11, 3.

D - Incorrect: Insertion rearranges the list in its existing slots. See assigned PDF 11, 11, 11, 3.

Primary note proof

Study-note section DSA37

Ordinary array merging needs a linear auxiliary buffer.

Study-note section DSA36

Quicksort normally rearranges in the same array but recursive storage can be linear in the worst case.

Study-note section DSA38

Building with bottom-up sinking is linear; repeated extraction gives O(n log n) worst time. Iterative sinking needs constant auxiliary storage. The method is generally unstable because long root-to-end exchanges can reorder equal keys. It is not uniquely the method for sorting by priorities: any comparison sort can compare a priority field. Heap extraction is its distinctive mechanism.

Study-note section DSA34

For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which

each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Data Structures & Algorithms practice questions · Question 31

Question 52 | Q00402

Which of the following is not a stable sorting algorithm?

A. Quick sort

B. Cocktail sort

C. Bubble sort

D. Merge sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Standard quicksort is unstable.

The DSA31 sorting table identifies quicksort as unstable. Cocktail and bubble sorts preserve equal-key order when swapping only strict adjacent inversions. Ordinary merge sort preserves equal-key order when ties are taken from the left run first. Under these usual implementations, quicksort is the single listed algorithm that does not guarantee stability.

Why each choice is right or wrong

A - Correct: Standard quicksort partition exchanges can move equal keys past one another and do not guarantee stability.

B - Incorrect: Cocktail sort is stable with strict adjacent swaps, as described in DSA32.

C - Incorrect: Bubble sort preserves equal-key order when it swaps only strict adjacent inversions.

D - Incorrect: Merge sort preserves equal-key order by taking equal elements from the left run first.

Primary note proof

Study-note section DSA32

> Early stopping makes a sorted input linear: one no-swap pass finishes. A fixed pass schedule stays quadratic. Strict swaps preserve equal-key order. Cocktail sort runs bubble passes in both directions and is stable with strict swaps.

Study-note section DSA37

> With left-first tie handling, equal keys retain order.

Study-note section DSA31

> stack usually log n average, n worst; unstable

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

> ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
> Algorithm:
> 1. Choose a pivot value (mostly the first element is taken as the pivot value)
> 2. Position the pivot element and partition the list so that:
> • the left part has items less than or equal to the pivot value
> • the right part has items greater than or equal to the pivot value
> 3. Recursively sort the left part
> 4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 4 | lines 2-8

> C. Bubble Sort
> The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through

the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Data Structures & Algorithms practice questions · Question 32

Question 53 | Q00403

Which specified sorting implementations avoid recursion? A is ordinary recursive quicksort; B is top-down recursive mergesort; C uses iterative heap sinking; D is bottom-up mergesort. Select all that apply.

A. quick sort

B. top-down merge sort

C. heap sort with iterative sinking

D. bottom up merge sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C, D: C. heap sort with iterative sinking; D. bottom up merge sort

Bottom-up mergesort processes run widths 1, 2, 4 and so on in loops, without recursive splitting. Heapsort can also restore heap order with iterative sinking, making the specified C implementation nonrecursive. A and B are explicitly the recursive forms described in the note. The question fixes implementations because algorithm names alone do not force either recursive or iterative control flow.

Why each choice is right or wrong

A - Incorrect: The specified quicksort recursively sorts its partition ranges.

B - Incorrect: The specified top-down mergesort recursively sorts halves.

C - Correct: Iterative heap sinking requires no recursive calls.

D - Correct: Bottom-up run merging proceeds iteratively.

Primary note proof

Study-note section DSA37

> Top-down mergesort recurses; bottom-up merges runs of widths 1,2,4,... iteratively.

Study-note section DSA38

> Iterative sinking needs constant auxiliary storage.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

> ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
> Algorithm:
> 1. Choose a pivot value (mostly the first element is taken as the pivot value)
> 2. Position the pivot element and partition the list so that:
> • the left part has items less than or equal to the pivot value
> • the right part has items greater than or equal to the pivot value
> 3. Recursively sort the left part
> 4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

> iv. Merge sort
> Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
> Algorithm:
> 1. Divide the array in to two halves.
> 2. Recursively sort the first n/2 items.
> 3. Recursively sort the last n/2 items.
> 4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 11 | lines 23-37

> iii. Heap Sort
> Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
> each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
> Algorithm:
> 1. Construct a binary tree
> • The root node corresponds to Data[0].
> • If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the

array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Data Structures & Algorithms practice questions · Question 33

Question 54 | Q00404

What is the worst case time complexity of a quick sort algorithm?

A. O(N)

B. O(N log N)

C. O(N2)

D. O(log N)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: O(N2)

The source states O(n^2) in the worst case. Repeated extreme pivots leave subproblems n-1,n-2,..., whose partition work sums quadratically. Evidence: assigned PDF pages 11.

Why each choice is right or wrong

A - Incorrect: Linear does not include repeated near-full partitions. See assigned PDF 11.

B - Incorrect: n log n is the expected, not worst, cost. See assigned PDF 11.

C - Correct: Unbalanced partitions give quadratic work. See assigned PDF 11.

D - Incorrect: Logarithmic describes neither partition scans nor their total. See assigned PDF 11.

Primary note proof

Study-note section DSA36

Choose a pivot, partition, then recursively sort the pieces. Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work. Expected n log n needs an input/random-pivot model. Random and median-of-three choices have different tradeoffs; neither is universally best.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Question sources

Data Structures & Algorithms practice questions · Question 34

Question 55 | Q00405

For a strictly increasing array of at least five distinct keys, which listed pivot rule is deterministic and avoids choosing an extreme key on the first partition? Interpret median-of-three as the median of the first, middle and last keys.

A. first element

B. last element

C. median-of-three partitioning

D. random element

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: C. median-of-three partitioning

On a strictly increasing array, the first key is the minimum and the last key is the maximum, so those deterministic endpoint rules choose extremes. The median of the first, middle and last keys is the middle-position value, which is neither extreme for the stated size. A random element is not a deterministic rule and can choose an endpoint. This criterion selects C without claiming a universally best practical pivot strategy.

Why each choice is right or wrong

A - Incorrect: The first key is the minimum on this input.

B - Incorrect: The last key is the maximum on this input.

C - Correct: The sample median is the middle-position value and is not extreme.

D - Incorrect: Random selection is not deterministic and may choose an extreme.

Primary note proof

Study-note section DSA36

Choose a pivot, partition, then recursively sort the pieces. Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work. Expected n log n needs an input/random-pivot model. Random and median-of-three choices have different tradeoffs; neither is universally best.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Question sources

Data Structures & Algorithms practice questions · Question 35

Question 56 | Q00406

What is the average running time of a quick sort algorithm?

A. O(N2)

B. O(N)

C. O(N log N)

D. O(log N)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer C: O(N log N)

Expected O(n log n) is explicitly given. Under ordinary nonadversarial/random-pivot assumptions this is the average runtime intended by the question. Evidence: assigned PDF pages 11.

Why each choice is right or wrong

A - Incorrect: Quadratic is the stated worst case. See assigned PDF 11.

B - Incorrect: Linear omits recursive partition levels. See assigned PDF 11.

C - Correct: Matches the stated expected n log n growth. See assigned PDF 11.

D - Incorrect: Logarithmic omits linear partition work. See assigned PDF 11.

Primary note proof

Study-note section DSA36

> Choose a pivot, partition, then recursively sort the pieces. Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work. Expected n log n needs an input/random-pivot model. Random and median-of-three choices have different tradeoffs; neither is universally best.

Study-note section DSA-S01

> Fix the input size before comparing cases. Best-case cost is the minimum resource use over legal inputs of that size; worst-case cost is the maximum. Average-case cost averages costs using a stated input distribution. These three input-case functions can each be described using upper, lower or tight asymptotic bounds; the case name does not select the bound notation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

> ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
> Algorithm:
> 1. Choose a pivot value (mostly the first element is taken as the pivot value)
> 2. Position the pivot element and partition the list so that:
> • the left part has items less than or equal to the pivot value
> • the right part has items greater than or equal to the pivot value
> 3. Recursively sort the left part
> 4. Recursively sort the right part

Question sources

Data Structures & Algorithms practice questions · Question 36

Question 57 | Q00407

Which option gives the most complete set of the listed situations in which linear search can be an appropriate direct method? Select the combined answer rather than only one valid situation.

A. When the list has only a few elements

B. When performing a single search in an unordered list

C. Used all the time

D. When the list has only a few elements and When performing a single search in an unordered list

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: Both short lists and a single unsorted-list query favor scanning.

Linear scanning can suit a short list because setup costs may dominate, and it can answer a single unsorted query without first building an ordered index or sorting the data. These are two supported use cases, not conditions that must occur simultaneously. A and B each give one true but incomplete situation. D combines both, while the universal claim that scanning should be used all the time is false.

Why each choice is right or wrong

A - Incorrect: True but incomplete: a short list is one supported case, but this omits the single unsorted query.

B - Incorrect: True but incomplete: a single unsorted query is supported, but this omits the short-list case.

C - Incorrect: Linear search is not preferable for every workload or available representation.

D - Correct: The combined answer includes both supported situations.

Primary note proof

Study-note section DSA28

> For a single unsorted query, sorting first is rarely justified. Small lists may favour scanning because setup matters. Sorting does not avoid examining all values for mean, variance or arbitrary parity counts.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first element
> • Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

Data Structures & Algorithms practice questions · Question 37

Question 58 | Q00408

What is the best case for linear search?

A. O(nlogn)

B. O(logn)

C. O(n)

D. O(1)

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: O(1)

An immediate match at the first list element takes one comparison. Treat complexity options as the tight order. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: n log n is not the immediate-match cost. See assigned PDF 3.

B - Incorrect: Logarithmic is not needed for the first inspected item. See assigned PDF 3.

C - Incorrect: Linear is the complete-scan order. See assigned PDF 3.

D - Correct: The first-item match requires constant work. See assigned PDF 3.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first
> element
> • Time is proportional to the size of input

(n) and we call this time complexity
O(n).

Question sources

Data Structures & Algorithms practice questions · Question 38

Question 59 | Q00409

What is the worst case for linear search?

A. O(nlogn)

B. O(logn)

C. O(n)

D. O(1)

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: O(n)

An absent target or a match at the final position requires inspecting n elements, giving tight linear worst-case growth. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: n log n is a loose bound, not the scan order. See assigned PDF 3.

B - Incorrect: Logarithmic cannot inspect all n elements. See assigned PDF 3.

C - Correct: A full scan is proportional to n. See assigned PDF 3.

D - Incorrect: Constant work describes immediate success, not worst case. See assigned PDF 3.

Primary note proof

Study-note section DSA28

Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List

• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Data Structures & Algorithms practice questions · Question 39

Question 60 | Q00410

What is the worst case complexity of binary search using recursion?

A. O(nlogn)

B. O(logn)

C. O(n)

D. O(n2)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer B: O(logn)

Each recursive step halves the candidate interval; after k steps its size is at most n/2^k. Reaching size one requires about log2 n steps. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: n log n overcounts a single search path. See assigned PDF 3.

B - Correct: Repeated halving gives logarithmic search depth. See assigned PDF 3.

C - Incorrect: Linear ignores the discarded half at every step. See assigned PDF 3.

D - Incorrect: Quadratic has no corresponding nested scan. See assigned PDF 3.

Primary note proof

Study-note section DSA29

On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Study-note section DSA29

Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal

loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Data Structures & Algorithms practice questions · Question 40

Question 61 | Q00411

For recursive binary search on a sorted random-access array of n distinct keys, each stored key is equally likely to be the successful target. With constant-time comparisons, what is the tightest listed average-time upper bound?

A. O(nlogn)

B. O(logn)

C. O(n)

D. O(n2)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer B: B. O(logn)

Binary search repeatedly halves the remaining interval. Under equally likely successful keys, only a limited number can be found at shallow decision-tree levels, while a constant fraction require logarithmic depth. Together with the logarithmic upper bound, this gives Theta(log n) average probes. The larger displayed upper bounds are valid but loose; the requested tightest listed bound is therefore O(log n).

Why each choice is right or wrong

A - Incorrect: n log n is a looser upper bound on logarithmic average work.

B - Correct: log n is the tight average order under the stated successful-target distribution.

C - Incorrect: Linear is a valid but looser upper bound.

D - Incorrect: Quadratic is a valid but much looser upper bound.

Primary note proof

Study-note section DSA-S01

> For successful binary search with all stored positions equally likely, average probes have order Theta(log n). The probe decisions form a binary tree: only a bounded number of positions occur in shallow levels, while a constant fraction require logarithmic depth. Interval halving also gives the logarithmic upper bound. A distribution concentrating on an immediate midpoint hit would give a different average.

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

Data Structures & Algorithms practice questions · Question 41

Question 62 | Q00412

Which task cannot directly use ordinary binary interval elimination under its stated conditions? The interval-union subtask in B uses a sorted array of interval endpoints; C has an ordered monotone good/bad boundary.

A. To find the lower/upper bound in an ordered sequence

B. Finding endpoint bounds in a sorted endpoint array while forming an interval union

C. Finding a monotone good/bad boundary during debugging

D. Directly searching an unordered list without sorting or an ordered index

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: D. Directly searching an unordered list without sorting or an ordered index

Binary search needs ordering that justifies rejecting one half of its candidate interval. Lower and upper bounds in an ordered sequence have that property. Searching sorted interval endpoints during an interval-union task also supplies an ordered subproblem, and a monotone debugging boundary permits the same halving logic. Direct searching in an unordered list supplies no such invariant, so a midpoint comparison cannot safely discard a half.

Why each choice is right or wrong

A - Incorrect: Boundary variants operate on the ordered candidate sequence.

B - Incorrect: The specified endpoint lookup is an ordered binary-search subtask.

C - Incorrect: The specified monotone good/bad boundary supports binary elimination.

D - Correct: An unordered list lacks the order needed to safely discard half after comparison.

Primary note proof

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Study-note section DSA29

> Binary debugging needs an ordered good/bad boundary, not arbitrary fault locations.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

Data Structures & Algorithms practice questions · Question 42

Question 63 | Q00413

Which standard linear structure is defined by a chain of nodes, each containing a payload and one successor link, with a null successor ending an ordinary noncircular chain?

A. linked list

B. Array

C. Binary tree

D. None of these

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. linked list

The payload stores an item and the successor link connects its node to the next item's node. Following those links produces a linear sequence ending at null, which is the ordinary singly linked-list representation. An array instead reaches equal-sized positions by address arithmetic, and a binary tree has a branching parent/child contract rather than this one-successor sequence contract. The linked-list option is therefore the matching standard name.

Why each choice is right or wrong

A - Correct: A linked list represents its sequence through payload-plus-successor nodes.

B - Incorrect: An array uses indexed positions rather than this successor-chain representation.

C - Incorrect: A binary tree specifies up to two child branches rather than the stated linear successor contract.

D - Incorrect: The linked-list option supplies the matching structure.

Primary note proof

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain.

Study-note section DSA02

> For equal-size elements, address(A[i]) = base + i × element_size under zero-based indexing.

Study-note section DSA18

> Binary means at most two children.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Data Structures & Algorithms practice questions · Question 43

Question 64 | Q00414

What is the time complexity to count the number of elements in the linked list?

A. O(1)

B. O(n)

C. O(logn)

D. None of the mentioned

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer B: O(n)

Without a stored count, follow next pointers from start to NULL and increment once per visited node. Exactly n node visits are needed. Evidence: assigned PDF pages 5.

Why each choice is right or wrong

A - Incorrect: Constant time requires an additional maintained count. See assigned PDF 5.

B - Correct: Traversal visits every node once. See assigned PDF 5.

C - Incorrect: No halving shortcut exists in the next-pointer chain. See assigned PDF 5.

D - Incorrect: B supplies the traversal order. See assigned PDF 5.

Primary note proof

Study-note section DSA08

To display or count nodes, follow next from head until null, processing each node. Counting is Theta(n) unless size is maintained. An array-linked list still follows links unless separately position-indexed.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 5 | lines 26-34

III. Displaying the list of Nodes
After adding nodes to the list, we need to display the list of nodes on the screen. Here are the steps required to display all nodes item in the list:
1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the next pointer of the node it is currently indicating.
5. Jump back to step 2.

Question sources

Data Structures & Algorithms practice questions · Question 44

Question 65 | Q00415

Linked lists are not suitable to for the implementation of?

A. Insertion sort

B. Radix sort

C. Polynomial manipulation

D. Binary search

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: Binary search loses its usual advantage on linked lists.

DSA29 explains that linked midpoint traversal loses logarithmic total time. That makes an ordinary linked list unsuitable for the usual efficient binary-search implementation, although an algorithm can still be written. The other tasks do not require constant-time midpoint access: insertion sort can relink nodes, polynomial terms can occupy nodes, and radix digit buckets can retain stable node sequences.

Why each choice is right or wrong

A - Incorrect: Insertion sorting can find an insertion location and relink nodes without requiring constant-time midpoint access.

B - Incorrect: Radix digit buckets can preserve node order across passes without a random-access midpoint operation.

C - Incorrect: Polynomial terms are explicitly listed as a linked-list application in DSA10.

D - Correct: Traversing to linked-list midpoints defeats the standard logarithmic total-time advantage of binary search.

Primary note proof

Study-note section DSA29

> Linked midpoint traversal loses logarithmic total time.

Study-note section DSA34

> Few inversions mean few shifts, favouring insertion for nearly sorted or small arrays. Shift only strictly larger keys for stability. A linked implementation saves array shifts but still must locate each insertion position.

Study-note section DSA10

> Lists suit changing sequences when references to insertion positions are already available. Finding a position is still linear. Lists can hold polynomial terms (coefficient, exponent), directory or block links, child lists in nonbinary trees, and hash chains. Browser history can use a doubly linked cursor. A playlist may be circular; airline seat lookup requires a specified seat-record model, not an automatic linked-list answer.

Study-note section DSA39

> Stable intermediate passes preserve the ordering already established by less-significant digits.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
> to the chain.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Data Structures & Algorithms practice questions · Question 45

Question 66 | Q00416

Which of these is an application of linked lists?

A. To implement file systems

B. For separate chaining in hash-tables

C. To implement non-binary trees

D. All of the mentioned

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: All listed applications.

DSA10 explicitly lists directory or block links, hash chains, and nonbinary-tree child lists as applications of linked nodes. These correspond to the three preceding options. Therefore All of the mentioned is the complete answer. The statement says linked lists can implement these components; it does not require every file system or tree representation to use them.

Why each choice is right or wrong

A - Incorrect: File-system directory or block links are valid uses, but this option omits the other supported applications.

B - Incorrect: Separate chaining uses linked bucket entries, but this is only one of the supported applications.

C - Incorrect: Child lists can represent nonbinary-tree children, alongside the other listed uses.

D - Correct: All three preceding options correspond to linked-record applications explicitly listed in DSA10.

Primary note proof

Study-note section DSA10

Lists suit changing sequences when references to insertion positions are already available. Finding a position is still linear. Lists can hold polynomial terms (coefficient, exponent), directory or block links, child lists in nonbinary trees, and hash chains. Browser history can use a doubly linked cursor. A playlist may be circular; airline seat lookup requires a specified seat-record model, not an automatic linked-list answer.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Data Structures & Algorithms practice questions · Question 46

Question 67 | Q00417

In linked list each node contain minimum of two fields. One field is data field to store the data second field is?

A. Pointer to character

B. Pointer to integer

C. Pointer to node

D. Node

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Pointer to node

The second minimum field points to the next node of the same node structure, alongside the payload. Evidence: assigned PDF pages 4.

Why each choice is right or wrong

A - Incorrect: A character pointer is not the next-node link unless the entire node type were a character, which it is not. See assigned PDF 4.

B - Incorrect: An integer pointer cannot stand for the described structured node link. See assigned PDF 4.

C - Correct: The link points to a node of the list. See assigned PDF 4.

D - Incorrect: The field is a pointer to a node, not a nested node stored by value. See assigned PDF 4.

Primary note proof

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
> to the chain.

Question sources

Data Structures & Algorithms practice questions · Question 47

Question 68 | Q00418

For repeated queries for the item at logical position k, which listed representation gives constant-time lookup? A-C maintain ordinary links and endpoints only; D additionally maintains a random-access array mapping each logical position to its node. Ignore the cost of maintaining this index during updates.

A. Singly linked list

B. Doubly linked list

C. Circular linked list

D. Array implementation with a separate logical-position-to-node index

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer D: D. Array implementation with a separate logical-position-to-node index

Ordinary singly, doubly and circular linked lists must follow links to reach a logical position, giving linear worst-case positional access. Storing nodes in an array alone does not change that. The explicit extra index in D is different: it maps each logical position directly to the corresponding node, so a random-access array lookup answers the query in constant time. Update costs are excluded by the stated workload.

Why each choice is right or wrong

A - Incorrect: Ordinary next-link traversal is linear in the worst case.

B - Incorrect: Using either endpoint still requires linear worst-case traversal.

C - Incorrect: Circular links do not supply a direct logical-position index.

D - Correct: The maintained position-to-node array permits direct indexed lookup.

Primary note proof

Study-note section DSA08

> To display or count nodes, follow next from head until null, processing each node. Counting is Theta(n) unless size is maintained. An array-linked list still follows links unless separately position-indexed.

Study-note section DSA02

> For equal-size elements, address(A[i]) = base + i × element_size under zero-based indexing. This arithmetic gives constant-time positional access in the usual RAM cost model. A row-major rectangular matrix uses base + (row × columns + column) × element_size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 26-34

III. Displaying the list of Nodes
After adding nodes to the list, we need to display the list of nodes on the screen. Here are the steps required to display all nodes item in the list:
1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the next pointer of the node it is currently indicating.
5. Jump back to step 2.

Question sources

Data Structures & Algorithms practice questions · Question 48

Question 69 | Q00419

Linked list is considered as an example of _____ type of memory allocation.

A. Dynamic

B. Static

C. Compile time

D. None of the mentioned

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Dynamic

Nodes are allocated with new as they are needed; the source explicitly calls linked-list stack memory dynamic. Evidence: assigned PDF pages 5, 6.

Why each choice is right or wrong

A - Correct: Node allocation occurs during execution. See assigned PDF 5, 6.

B - Incorrect: Static allocation is the contrasting array implementation. See assigned PDF 5, 6.

C - Incorrect: Compile-time allocation does not describe new-node creation. See assigned PDF 5, 6.

D - Incorrect: Dynamic is explicitly supported. See assigned PDF 5, 6.

Primary note proof

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Study-note section DSA03

> In C++, match new with delete and new[] with delete[].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 5 | lines 3-8

> While adding a node to a list, the chain needs to be rearranged properly so that the each nodes are connected to each other. It is possible to add a node anywhere in the chain (at the start, middle or end). Let’s consider adding a node at the end of the list:
> Step 1: First we need to reserve a space for the new node using the keyword new.
> Node *temp = new *Node; // Reserving a space for temporary pointer which points to new node;

Data Structure and Algorithms module | PDF page 6 | lines 43-45

> II. Linked list (dynamic memory)
> The linked list implementation of stack is adding items at the end of the chain (Push item) and deleting the end node of the list (Pop item).

Question sources

Data Structures & Algorithms practice questions · Question 49

Question 70 | Q00420

In Linked List implementation, a node carries information regarding

A. Data

B. Link

C. Data and Link

D. None of the mentioned

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Data and Link

A node combines its payload with a next-node pointer. Neither field alone represents the complete linked-list node. Evidence: assigned PDF pages 4.

Why each choice is right or wrong

A - Incorrect: Data alone omits the next link. See assigned PDF 4.

B - Incorrect: A link alone omits the payload. See assigned PDF 4.

C - Correct: Both data and link are explicitly required. See assigned PDF 4.

D - Incorrect: C matches the stated two-field structure. See assigned PDF 4.

Primary note proof

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
> to the chain.

Question sources

Data Structures & Algorithms practice questions · Question 50

Question 71 | Q00421

Process of inserting an element in stack is called _____.

A. Create

B. Push

C. Evaluation

D. Pop

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: Push

The module defines push as adding an element to the stack top. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: Create would initialize a structure, not name its insertion operation. See assigned PDF 6.

B - Correct: Push adds an element. See assigned PDF 6.

C - Incorrect: Evaluation is an application rather than insertion. See assigned PDF 6.

D - Incorrect: Pop removes an element. See assigned PDF 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Data Structures & Algorithms practice questions · Question 51

Question 72 | Q00422

In a stack, if a user tries to remove an element from empty stack it is called _____.

A. Underflow

B. Empty collection

C. Overflow

D. Garbage Collection

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Underflow

The stack pop code first tests empty at top==0. Removing from an empty structure is the underflow condition, explicitly named for dequeue and applied here to the same empty-removal failure. Evidence: assigned PDF pages 6, 7.

Why each choice is right or wrong

A - Correct: Underflow denotes attempting removal without an available element. See assigned PDF 6, 7.

B - Incorrect: Empty collection describes the state rather than the operation failure. See assigned PDF 6, 7.

C - Incorrect: Overflow concerns adding to a full structure. See assigned PDF 6, 7.

D - Incorrect: Garbage collection concerns reclaiming storage, not empty pop. See assigned PDF 6, 7.

Primary note proof

Study-note section DSA11

> Overflow is an attempted insertion into full storage; underflow is an attempted removal from empty storage. Merely checking isEmpty does not underflow.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

> I. Array implementation (static memory).
> Push(n): Adding item at top
> top is the current top of stack and n is its
> maximum size
> Begin
> if (top == n) then stack is full;
> else

```
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;
```

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Data Structures & Algorithms practice questions · Question 52

Question 73 | Q00423

Process of removing an element from stack is called _____

A. Create

B. Push

C. Evaluation

D. Pop

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: Pop

The source names removing an element Pop and adding one Push. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: Create does not remove the stack top. See assigned PDF 6.

B - Incorrect: Push adds rather than removes. See assigned PDF 6.

C - Incorrect: Evaluation uses stack operations but is not their deletion name. See assigned PDF 6.

D - Correct: Pop removes the current top. See assigned PDF 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Data Structures & Algorithms practice questions · Question 53

Question 74 | Q00424

A serial-storage interface restricts insertion and removal to the same current end: an item must be removed before any item stored earlier beneath it can be removed. Which ADT matches this restricted access policy?

A. Heap

B. Binary Tree

C. Array

D. Stack

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: D. Stack

The restriction makes the latest inserted item the next removable item. That is the stack's LIFO policy, with both push and pop acting at the top. Serial access alone would not establish this rule, so the same-end removal restriction is stated explicitly. A heap selects by priority, a binary tree supplies branching relationships, and an ordinary array permits indexed access rather than imposing this removal policy.

Why each choice is right or wrong

A - Incorrect: A heap selects an extreme priority rather than the latest insertion.

B - Incorrect: A binary tree supplies hierarchy rather than this same-end policy.

C - Incorrect: An array does not inherently impose this removal order.

D - Correct: A stack's push/pop policy matches the specified LIFO restriction.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA45

> ADTs do not force one representation, and serial-access memory is not inherently LIFO.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> "A stack is an ordered list in which all insertions and deletions are made at one end, called the top". Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Data Structures & Algorithms practice questions · Question 54

Question 75 | Q00425

A singly linked queue maintains only its front pointer and owns all its nodes. Emptying means immediately visiting and freeing every node. Which option gives the complete set of listed operations with tight worst-case linear time for n nodes?

A. Insertion

B. Deletion

C. To empty a queue

D. Both Insertion and To empty a queue

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer D: D. Both Insertion and To empty a queue

Enqueue must traverse from the front to find the tail because no rear pointer is maintained. Emptying the owned queue must also visit and release all n nodes under the specified reclamation contract. Deleting one front node requires only a constant number of link and deallocation operations. Therefore both insertion and complete emptying are linear, and the aggregate option D is the complete selection.

Why each choice is right or wrong

A - Incorrect: Insertion is linear, but this omits the also-linear emptying operation.

B - Incorrect: A single front deletion is constant time.

C - Incorrect: Emptying is linear, but this omits insertion.

D - Correct: Both insertion and immediate full-node cleanup are linear.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Study-note section DSA09

> Deleting the tail of a singly linked list requires its predecessor, even when tail itself is known. Handle empty and single-node lists separately; after removing the only node, set both head and tail to null. To remove a known successor, bypass it before deallocation. Clearing an owned n-node chain visits and releases every node.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 5 | lines 13-24

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.
if (start == NULL) // If list is empty, then new node become start node
start = temp;
else {
temp2 = start; // list not empty! Thus temp2 starts from first node
while (temp2->next!= NULL) { // till the current last node found navigates forward
temp2 = temp2->next; // Move to next link in chain
}
temp2->next = temp; // new node (temp) appended at the end of the list

Data Structure and Algorithms module | PDF page 8 | lines 35-37

III. Linked list implementation of enqueue and dequeue operations
Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

Data Structure and Algorithms module | PDF page 5 | lines 35-41

IV. Deleting a node from the list
When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the delete instruction:
delete temp; // Release the memory pointed to by temp
However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. We can delete nodes at the start, middle or end of the list. Here let’s see the steps to delete at the end.

Question sources

Data Structures & Algorithms practice questions · Question 55

Question 76 | Q00426

A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a ?

A. Queue

B. Stack

C. Tree

D. Linked list

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Queue

The queue definition makes insertion rear-only and deletion front-only. A linked list can implement that policy, but does not by itself impose the two restrictions. Evidence: assigned PDF pages 7, 6, 4.

Why each choice is right or wrong

A - Correct: Matches the defined queue interface. See assigned PDF 7, 6, 4.

B - Incorrect: Stack uses one common endpoint for both operations. See assigned PDF 7, 6, 4.

C - Incorrect: Tree operations are not front/rear list operations. See assigned PDF 7, 6, 4.

D - Incorrect: A generic linked list permits operations beyond queue restrictions. See assigned PDF 7, 6, 4.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem

• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Data Structures & Algorithms practice questions · Question 56

Question 77 | Q00427

In Breadth First Search of Graph, which of the following data structure is used?

A. Stack

B. Queue

C. Linked list

D. None of the mentioned

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: Queue

DSA41 repeatedly dequeues a vertex and enqueues undiscovered neighbors. FIFO service preserves the order of distance layers. Queue is therefore the standard BFS frontier structure. A linked list may store the queue but does not alone define this ordering; a stack chooses latest pending work and instead supports the depth-first discipline.

Why each choice is right or wrong

A - Incorrect: A stack selects the latest pending vertex and does not maintain breadth-first distance layers.

B - Correct: A FIFO queue processes earlier discoveries before later layers, matching standard breadth-first search.

C - Incorrect: Linked nodes can store a queue, but linked-list structure alone specifies no BFS frontier policy.

D - Incorrect: Queue is offered and directly matches the algorithm given in DSA41.

Primary note proof

Study-note section DSA41

> Mark the source when enqueuing it. Repeatedly dequeue a vertex, examine its neighbours, and enqueue each unvisited neighbour while marking it. Store predecessor[v] when discovering v if a path must be reconstructed. FIFO ensures vertices at distance d are explored before later distance layers.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Data Structures & Algorithms practice questions · Question 57

Question 78 | Q00428

A data structure in which elements can be inserted or deleted at/from both the ends but not in the middle is?

A. Queue

B. Circular queue

C. Dequeue

D. Priority queue

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Dequeue

Interpret the option spelling Dequeue as deque, the double-ended queue defined on page 8. That structure allows insertion and deletion at either end. Evidence: assigned PDF pages 8, 7.

Why each choice is right or wrong

A - Incorrect: An ordinary queue restricts insertion and deletion to different fixed ends. See assigned PDF 8, 7.

B - Incorrect: Circular storage changes reuse, not the ordinary queue interface. See assigned PDF 8, 7.

C - Correct: Deque is explicitly double-ended; the option has a terminology typo. See assigned PDF 8, 7.

D - Incorrect: Priority queue chooses service by key, not arbitrary endpoint operations. See assigned PDF 8, 7.

Primary note proof

Study-note section DSA16

> A deque permits both-end insertion/removal without guaranteeing middle access. Endpoint-linked operations are constant but middle indexing is linear. A circular-array deque may support indexed access. Dequeue names removal; deque names the structure.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

> Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
> Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Data Structures & Algorithms practice questions · Question 58

Question 79 | Q00429

Queues serve major role in

A. Simulation of recursion

B. Simulation of arbitrary linked list

C. Simulation of limited resource allocation

D. All of the mentioned

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Simulation of limited resource allocation

The module lists print jobs, disk requests and waiting lines: all allocate a limited resource among waiting requests. Recursion is expressly a stack use, so all is false. Evidence: assigned PDF pages 8, 6.

Why each choice is right or wrong

A - Incorrect: Recursion is listed under stack applications. See assigned PDF 8, 6.

B - Incorrect: Arbitrary linked-list simulation is not a stated queue role. See assigned PDF 8, 6.

C - Correct: Waiting requests for shared resources fit the listed applications. See assigned PDF 8, 6.

D - Incorrect: A is false, so the conjunction cannot hold. See assigned PDF 8, 6.

Primary note proof

Study-note section DSA14

> FIFO suits equal-priority printer jobs, shared-resource waiting, asynchronous buffers and level-order exploration. A play sequence can also use a queue. Distinguish airline seat records from waiting requests; an application name alone does not force one representation.

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 41-48

> Application of Queue
> Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
> • Print server- maintains a queue of print jobs
> • Disk Driver- maintains a queue of disk input/output requests

• Task scheduler in multiprocessing system- maintains priority queues of processes
• Telephone calls in a busy environment -maintains a queue of telephone calls
• Simulation of waiting line- models real life queues (e.g. supermarkets checkout, customer service line (Bank, custom, airport etc))

Data Structure and Algorithms module | PDF page 6 | lines 46-52

Applications of Stack
• It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
• Infix to Postfix Transformation
• It is useful during the execution of recursive programs
• A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
• A stack (memory stack) can be used in function calls including recursion.

Question sources

Data Structures & Algorithms practice questions · Question 59

Question 80 | Q00430

Which listed service discipline is a stack policy rather than FIFO, a circular FIFO representation, or priority-based queue service? Interpret B as insertion and removal at the same top end.

A. Ordinary queue

B. Single-ended insertion and removal at the same top (LIFO)

C. Circular queue

D. Priority queue

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: B. Single-ended insertion and removal at the same top (LIFO)

An ordinary queue serves front elements after rear insertion, while a circular queue changes how its FIFO storage slots are reused. A priority queue chooses according to a priority key. Inserting and removing at the same top end instead gives LIFO stack service. Explicitly defining the formerly vague single-ended phrase makes this a policy distinction rather than an unsupported claim that a particular phrase is forbidden terminology.

Why each choice is right or wrong

A - Incorrect: Ordinary queue service is FIFO.

B - Correct: Same-top insertion/removal is LIFO stack service.

C - Incorrect: Circular storage can implement FIFO queue service.

D - Incorrect: Priority queue service selects by an extreme priority key.

Primary note proof

Study-note section DSA11

Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA14

An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Study-note section DSA17

A priority queue returns the smallest or largest priority key under its convention. Urgent patients may precede earlier arrivals. Insertions and extractions can interleave. Binary heaps support constant peek and logarithmic insert/extract.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 7 | lines 24-26

II. Circular array implementation of Queue Operations
To overcome the above drawback of simple array implementation a circular array can be used, in
which freed spaces are re-used to store data.

Data Structure and Algorithms module | PDF page 8 | lines 40-45

Priority Queue:- is a queue where each data has an associated key provided at insertion time.
Application of Queue
Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
• Print server- maintains a queue of print jobs
• Disk Driver- maintains a queue of disk input/output requests
• Task scheduler in multiprocessing system- maintains priority queues of processes

Question sources

Data Structures & Algorithms practice questions · Question 60

Question 81 | Q00543

Which claims contradict the ordinary unique-key map contract? Select all that apply.

A. Allow duplicate keys

B. Cannot be traversed

C. Allow duplicate values

D. None of the above

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A, B: A. Allow duplicate keys; B. Cannot be traversed

A map associates each unique key with a value. Different keys may share the same value, and its entries can be traversed even when a particular iteration order is not promised. Thus allowing duplicate keys and forbidding all traversal both contradict the contract. Allowing duplicate values does not contradict it. Since contradictory claims exist, the none-of-the-above option is not selected.

Why each choice is right or wrong

A - Correct: Duplicate keys contradict the unique-key map contract.

B - Correct: Map entries can be traversed, so this prohibition is false.

C - Incorrect: Different keys may share values, making this a valid property.

D - Incorrect: A and B do contradict the contract.

Primary note proof

Study-note section DSA27

> A map associates unique keys with values; different keys may share values, and entries can be traversed. A set forbids duplicate elements. Null and iteration-order policies depend on implementation. Lists preserve positions and generally allow duplicates; Java LinkedList also permits null.

Question sources

Computer Science model exam · 2015 · Question 12

Computer Science model exam · 2015 · Question 113

Question 82 | Q00591

Consider a set implementation that permits one null element and does not promise an iteration order. Which statement about this implementation is incorrect?

A. Unordered collection

B. Duplicates are not allowed

C. null is not allowed

D. None of the above

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: C. null is not allowed

All sets forbid duplicate elements, but null acceptance and iteration order depend on the implementation. The implementation explicitly permits null and promises no iteration order. Under those policies, calling its collection unordered is consistent and rejecting duplicates remains correct. The claim that null is not allowed is the one contradiction. The none-of-the-above alternative fails because that contradictory claim is present.

Why each choice is right or wrong

A - Incorrect: No iteration order is promised under the specified policy.

B - Incorrect: Rejecting duplicate elements is the set contract.

C - Correct: The stated implementation permits null, contradicting this prohibition.

D - Incorrect: C is incorrect, so none-of-the-above fails.

Primary note proof

Study-note section DSA27

> A map associates unique keys with values; different keys may share values, and entries can be traversed. A set forbids duplicate elements. Null and iteration-order policies depend on implementation. Lists preserve positions and generally allow duplicates; Java LinkedList also permits null.

Question sources

Computer Science model exam · 2015 · Question 60

Question 83 | Q00593

Which one of the following is correct about list data structure?

A. Unordered collection

B. Duplicates are not allowed

C. null is not allowed

D. None of the above

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer D: None of the above under the ordinary list contract.

DSA27 says lists preserve positions and generally allow duplicates; it also gives Java LinkedList as a list permitting null. Therefore Unordered collection and Duplicates are not allowed misdescribe the ordinary list contract. The universal null prohibition is refuted by the supplied example. None of those statements is a generally correct list property, so D is supported.

Why each choice is right or wrong

A - Incorrect: Lists preserve positions, so calling the ordinary list contract unordered confuses positional order with sortedness.

B - Incorrect: Ordinary lists allow repeated elements; duplicate rejection belongs to sets rather than the general list contract.

C - Incorrect: Java LinkedList explicitly permits null, refuting a universal null prohibition for list structures.

D - Correct: None of the preceding statements describes a generally valid property of the ordinary list contract.

Primary note proof

Study-note section DSA27

> A map associates unique keys with values; different keys may share values, and entries can be traversed. A set forbids duplicate elements. Null and iteration-order policies depend on implementation. Lists preserve positions and generally allow duplicates; Java LinkedList also permits null.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and

pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Computer Science model exam · 2015 · Question 62

Question 84 | Q00711

In a binary search tree that stores distinct keys with strict left-smaller/right-larger ordering, which statements are true? Select all that apply.

A. Every node has at most two children.

B. The value on left sub tree of the binary search tree can be equal to the root node value.

C. Every node stores a unique key value.

D. All of the above

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A, C: A. Every node has at most two children.; C. Every node stores a unique key value.

Binary limits a node to at most two children; it does not require every node to have exactly two. The specified distinct-key model also requires uniqueness. Strict subtree ordering places every left-subtree key below its ancestor, so an equal key cannot appear there. A and C consequently hold under the stated model, while B and the all-of-the-above aggregate do not.

Why each choice is right or wrong

A - Correct: Binary means at most two children per node.

B - Incorrect: An equal key violates the specified strict left-subtree ordering.

C - Correct: Distinct keys mean every stored key value is unique.

D - Incorrect: B is false, so not all preceding statements hold.

Primary note proof

Study-note section DSA18

Binary means at most two children. Full means each node has zero or two children. Complete means all earlier levels are filled and the last fills from left. Perfect means all leaves share a level and every internal node has two children. Balanced means height stays logarithmic under a stated rule; it does not require a perfect shape.

Study-note section DSA19

With distinct keys, every key in a node's left subtree is smaller and every key in its right subtree is larger. Immediate-child comparisons alone are insufficient: root 10, left child 5, and 5's right child 12 violate the root's restriction. Duplicates require a separate count or placement policy.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science model exam 2 · Question 52

Question 85 | Q00712

The operation for adding an item to a stack is named to be___.

A. Push

B. Insert

C. Add

D. enqueue

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Push

The module names the stack addition operation Push. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Correct: Push is the defined stack insertion term. See assigned PDF 6.

B - Incorrect: Insert is a generic verb, not the given operation name. See assigned PDF 6.

C - Incorrect: Add is descriptive, not the stack-specific name. See assigned PDF 6.

D - Incorrect: Enqueue is the queue insertion name. See assigned PDF 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science model exam 2 · Question 53

Question 86 | Q00713

_____Which operation in stack could result underflow?

A. Push();

B. Is_empty();

C. Pop();

D. B and C

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Pop();

Only an attempted pop removes an element. When top==0 no element exists, so this removal attempt underflows; checking emptiness does not consume an element. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: Push can encounter a full stack, not empty-removal underflow. See assigned PDF 6, 6.

B - Incorrect: Is_empty only checks state. See assigned PDF 6, 6.

C - Correct: Pop attempts removal and can find the stack empty. See assigned PDF 6, 6.

D - Incorrect: B does not underflow, so B and C is false. See assigned PDF 6, 6.

Primary note proof

Study-note section DSA11

> Overflow is an attempted insertion into full storage; underflow is an attempted removal from empty storage. Merely checking isEmpty does not underflow.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

> I. Array implementation (static memory).
> Push(n): Adding item at top
> top is the current top of stack and n is its
> maximum size
> Begin
> if (top == n) then stack is full;
> else
> top := top+1;
> stack[top] := item;
> end:

```
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;
```

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science model exam 2 · Question 54

Question 87 | Q00714

During ordinary lookup in a binary search tree with distinct keys and strict left-smaller/right-larger ordering, which rule correctly chooses the next subtree or recognizes a match?

A. If the searched key is less than the root, go to left.

B. If the searched key is larger than the root, go to left.

C. If the searched key is equal to the root, go to left.

D. None.

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. If the searched key is less than the root, go to left.

The BST invariant puts every key in the current node’s left subtree below that node’s key and every right-subtree key above it. A target smaller than the current key must therefore be sought on the left. A larger target belongs on the right, while equality finishes the lookup at the current node. This directional rule relies on BST ordering, not merely on having a binary shape.

Why each choice is right or wrong

A - Correct: A smaller target follows the left subtree under the BST invariant.

B - Incorrect: A larger target follows right, not left.

C - Incorrect: Equality is a successful match rather than a reason to continue left.

D - Incorrect: A supplies a valid search rule.

Primary note proof

Study-note section DSA19

> With distinct keys, every key in a node's left subtree is smaller and every key in its right subtree is larger. Immediate-child comparisons alone are insufficient: root 10, left child 5, and 5's right child 12 violate the root's restriction. Duplicates require a separate count or placement policy.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

> • Binary tree: a tree in which each node has at most two children called left child and right child.
> • Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
> satisfies the following.
> o Every node has a key and no two elements have the same key.
> o The keys in the right subtree are larger than the keys in the root.
> o The keys in the left subtree are smaller than the keys in the root.
> o The left and the right subtrees are also binary search trees.

Question sources

Computer Science model exam 2 · Question 55

Question 88 | Q00715

Which key word is used to de-allocate the space taken by a node?

A. delete

B. alloc()

C. del()

D. A and B

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: delete

The deletion section explicitly uses delete temp to release node storage. Evidence: assigned PDF pages 5.

Why each choice is right or wrong

A - Correct: delete releases the pointed-to memory. See assigned PDF 5.

B - Incorrect: alloc is not the given C++ deallocation keyword. See assigned PDF 5.

C - Incorrect: del is not the module keyword. See assigned PDF 5.

D - Incorrect: B is not the prescribed deallocation instruction. See assigned PDF 5.

Primary note proof

Study-note section DSA03

> A C struct groups named fields, including arrays and pointers. A singly linked node needs payload plus a same-type link. C fields cannot have initializers inside the type declaration; modern C++ permits default member initializers. In C++, match new with delete and new[] with delete[]. Repair links before freeing a node; never dereference it afterwards.

Study-note section DSA09

> Prepend with new.next=head; head=new. To insert after a known predecessor p, set new.next=p.next before p.next=new. Appending with only head requires a tail traversal; maintaining tail avoids it. Delete the head by saving old=head, advancing head=head.next and releasing old in C++.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 5 | lines 35-41

> IV. Deleting a node from the list
> When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the delete instruction:
> delete temp; // Release the memory pointed to by temp
> However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. We can delete nodes at the start, middle or end of the list. Here let’s see the steps to delete at the end.

Question sources

Computer Science model exam 2 · Question 56

Question 89 | Q00716

Which one is easier to perform?

A. Deleting a leaf node

B. Deleting internal node

C. deleting root node

D. A and B

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: Deleting a leaf node

The module calls leaf deletion simple and straightforward: detach its parent link. Internal and root deletion can need child promotion or copying. Evidence: assigned PDF pages 10.

Why each choice is right or wrong

A - Correct: A leaf has no descendant subtree to reconnect. See assigned PDF 10.

B - Incorrect: An internal node can require preserving one or two child subtrees. See assigned PDF 10.

C - Incorrect: Root deletion may require choosing and reconnecting a replacement. See assigned PDF 10.

D - Incorrect: Internal deletion is not always as simple as leaf deletion. See assigned PDF 10.

Primary note proof

Study-note section DSA21

> A leaf can be detached by clearing its parent link. For one child, make the parent bypass the node and point to the child. For two children, copy the inorder predecessor (left-subtree maximum) or successor (right-subtree minimum), then delete that replacement from its original location. That second deletion has at most one child.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 10 | lines 5-19

> V. Deletion
> To delete a node (whose Num value is N) from binary search tree (whose root node is pointed by RootNodePtr), four cases should be considered. When a node is deleted the definition of binary search tree should be preserved.
> Case 1: Deleting a leaf node (a node having no child): it is simple and straightforward operation. Delete the leaf node and make the pointer of the deleted node parent NULL (right or left child pointer based on whether the deleted child is left child or right child)
> Case 2: Deleting a node having only one child

Case 3: Deleting a node having two children
Case 4: Deleting the root node
Note: For case 2, 3 and 4 there are two approaches
Approach 1: Deletion by copying
• Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
• Delete the Copied Node

Question sources

Computer Science model exam 2 · Question 57

Question 90 | Q00717

Algorithms must terminate after a finite number of steps; What is this characteristics called__

A. Unambiguous

B. Feasibility

C. Finiteness

D. Independent

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Finiteness

The algorithm criteria expressly name termination after a finite number of steps Finiteness. Evidence: assigned PDF pages 2.

Why each choice is right or wrong

A - Incorrect: Unambiguous describes definiteness of each instruction. See assigned PDF 2.

B - Incorrect: Feasibility is not the term attached to termination here. See assigned PDF 2.

C - Correct: Finiteness is the named finite-termination criterion. See assigned PDF 2.

D - Incorrect: Independent does not name finite termination. See assigned PDF 2.

Primary note proof

Study-note section DSA04

Finiteness means termination after finitely many steps for every valid input. Definiteness means each instruction has one interpretation; effectiveness means each step is executable. Correctness means meeting the stated result for every valid input. A finite program can still return the wrong answer.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 26-32

> Algorithm: An algorithm is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:
> • Input:there are zero or more quantities which are externally supplied;
> • Output: at least one quantity is produced;
> • Definiteness: each instruction must be clear and unambiguous;
> • Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Question sources

Computer Science model exam 2 · Question 58

Question 91 | Q00902

Which listed structures are linear data structures in the course classification? Select all that apply.

A. Tree

B. Array

C. Queue

D. Linked list

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B, C, D: B. Array; C. Queue; D. Linked list

Arrays, queues and linked lists arrange their items as linear sequences in this classification. A tree instead models hierarchical branching and is classified as nonlinear. Thus three offered structures meet the question’s category. The stray extraction characters before Queue are removed so the option states the structure’s name clearly, and the select-all instruction accounts for every valid member of the category.

Why each choice is right or wrong

A - Incorrect: A tree is hierarchical rather than linear.

B - Correct: An array is a linear structure.

C - Correct: A queue is a linear structure.

D - Correct: A linked list is a linear structure.

Primary note proof

Study-note section DSA01

An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 15-18

a. Linear data structure: data items stores in sequence
Example: Array, stack and Queue,
b. Non-Linear data structures : in which there is order of data items
Example: Trees and Graphs

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Computer Science Exit Exam · 2018 · Question 15

Question 92 | Q00931

Which abbreviations describe stack service order? Select all that apply.

A. LIFO

B. FIFO

C. FILO

D. FCFS

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A, C: A. LIFO; C. FILO

A stack removes the latest inserted item first, described as last in, first out or LIFO. The equivalent view says that the first inserted item leaves last, described as first in, last out or FILO. Both abbreviations name the same policy. FIFO and FCFS instead serve earlier arrivals before later ones and describe the ordinary queue discipline rather than the stack discipline.

Why each choice is right or wrong

A - Correct: LIFO means last in, first out.

B - Incorrect: FIFO is the ordinary queue discipline.

C - Correct: FILO is equivalent to LIFO.

D - Incorrect: FCFS serves arrivals in queue order.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA14

> This is FIFO or first-come, first-served.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science Exit Exam · 2018 · Question 44

Question 93 | Q00947

In singly linked list, the node contains

A. Data and index

B. Data and one pointer (the next node)

C. Only data

D. Data and two pointers

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: Data and one pointer (the next node)

The singly linked node contains a data payload and a single pointer to the next node. Evidence: assigned PDF pages 4.

Why each choice is right or wrong

A - Incorrect: An index is not the described next-pointer field. See assigned PDF 4.

B - Correct: Both required components are present. See assigned PDF 4.

C - Incorrect: Payload alone cannot connect the chain. See assigned PDF 4.

D - Incorrect: Two pointers describe a different representation, not the supplied single-link node. See assigned PDF 4.

Primary note proof

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various

operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Computer Science Exit Exam · 2018 · Question 60

Question 94 | Q00951

Which data structure follows LIFO principle?

A. Graphs

B. Queues

C. Stacks

D. Trees

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Stacks

Stack removes its most recently inserted top, exactly LIFO. Evidence: assigned PDF pages 6, 7, 2.

Why each choice is right or wrong

A - Incorrect: A graph structure does not prescribe a last-in deletion order. See assigned PDF 6, 7, 2.

B - Incorrect: Queue is FIFO. See assigned PDF 6, 7, 2.

C - Correct: Stack is expressly LIFO. See assigned PDF 6, 7, 2.

D - Incorrect: A tree hierarchy does not impose stack order. See assigned PDF 6, 7, 2.

Primary note proof

Study-note section DSA11

Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some

useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 7 | lines 10-16

"A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT". Queues are sometimes referred to as First in First out (FIFO) lists.
The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 2 | lines 15-18

a. Linear data structure: data items stores in sequence
Example: Array, stack and Queue,
b. Non-Linear data structures : in which there is order of data items
Example: Trees and Graphs

Question sources

Computer Science Exit Exam · 2018 · Question 64

Question 95 | Q00955

The time complexity for binary search is

A. O(n*log(n)2)

B. O(n2)

C. ¢.
O(n)

D. O(logn)

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer D: O(logn)

The source states search time proportional to log2 n; repeated midpoint selection halves the remaining ordered interval. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: n times a logarithmic factor is not the tight single-search bound. See assigned PDF 3.

B - Incorrect: Quadratic does not describe a halving path. See assigned PDF 3.

C - Incorrect: Linear is sequential-search growth. See assigned PDF 3.

D - Correct: Logarithmic matches the binary-search statement. See assigned PDF 3.

Primary note proof

Study-note section DSA29

Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Study-note section DSA06

For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science Exit Exam · 2018 · Question 68

Question 96 | Q00974

Which sorting algorithm compares each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swaps them; if at least one swaps has been done, repeat step 1.

A. Insertion sort

B. Bubble sort

C. .

Selection sort

D. Shell sort

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer B: Bubble sort

The stem describes complete adjacent-pair passes and repeats if swaps occur. That is bubble sort with a no-swap stopping test. Evidence: assigned PDF pages 4, 3.

Why each choice is right or wrong

A - Incorrect: Insertion places a new item in the sorted prefix. See assigned PDF 4, 3, 3.

B - Correct: Bubble compares and exchanges adjacent out-of-order entries across passes. See assigned PDF 4, 3, 3.

C - Incorrect: Selection searches for a minimum rather than exchanging every reversed adjacent pair. See assigned PDF 4, 3, 3.

D - Incorrect: Shell first uses gap-separated subsequences, unlike these adjacent passes. See assigned PDF 4, 3, 3.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right. The largest remaining value moves to the final unsorted position after one left-to-right pass. Shrink the next pass by one position. Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

> C. Bubble Sort
> The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
> the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.

• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Question sources

Computer Science Exit Exam · 2018 · Question 87

Question 97 | Q00978

Among the following one is NOT the application of queue in the real world.

A. It is used in operating systems for handling interrupts.

B. It is used to maintain the play list in media players in order

C. .
It is used in Depth First search

D. It is widely used as waiting lists for a single shared resource

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer C: Depth-first search uses a stack rather than the ordinary FIFO queue.

DSA14 gives FIFO waiting, asynchronous buffers, and play sequences as queue uses. DSA42 explicitly contrasts DFS stack or recursion with a queue, which would change the discipline. Pending interrupts can be buffered for processing; this does not mean every interrupt policy must be FIFO. Standard depth-first search is the listed application that requires latest-pending-work service instead.

Why each choice is right or wrong

A - Incorrect: Pending interrupt work can be queued for handling; the claim does not require every interrupt policy to be FIFO.

B - Incorrect: A queue can preserve a play sequence in arrival order, as noted in DSA14.

C - Correct: Standard DFS uses a stack or recursion to finish a branch before returning, not a FIFO frontier.

D - Incorrect: First-come shared-resource waiting is a direct FIFO queue application described in DSA14.

Primary note proof

Study-note section DSA-S02

> An interrupt service routine may need to record work quickly and defer its completion. Operating systems can queue that deferred work for later processing; Windows deferred procedure calls are one concrete example. This is a queue application, but it does not imply that every interrupt system uses a single FIFO service policy.

Study-note section DSA14

> FIFO suits equal-priority printer jobs, shared-resource waiting, asynchronous buffers and level-order exploration. A play sequence can also use a queue. Distinguish airline seat records from waiting requests; an application name alone does not force one representation.

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 41-48

> Application of Queue
> Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
> • Print server- maintains a queue of print jobs
> • Disk Driver- maintains a queue of disk input/output requests
> • Task scheduler in multiprocessing system- maintains priority queues of processes
> • Telephone calls in a busy environment -maintains a queue of telephone calls

• Simulation of waiting line- models real life queues (e.g. supermarkets checkout, customer service line (Bank, custom, airport etc))

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Question sources

Computer Science Exit Exam · 2018 · Question 91

Question 98 | Q00991

An airline has a fixed finite seat layout, and each seat number maps directly to a dense integer position. Which listed representation gives constant-time lookup of that seat's reservation record under the usual RAM model?

A. Array

B. Linked List

C. Queue

D. Stack

Hint

Fix the input case and cost model, then count dominant work or simultaneously live storage.

Answer A: A. Array

The explicit seat-number mapping turns each lookup into an indexed-position query over a fixed finite layout. An array can compute the corresponding element address directly, giving constant-time access in the note's RAM model. An ordinary linked list must traverse to a position, while queues and stacks expose service endpoints rather than arbitrary seat lookup. This models seat records themselves, rather than a queue of waiting reservation requests.

Why each choice is right or wrong

A - Correct: Direct seat-position indexing gives constant-time array lookup.

B - Incorrect: An ordinary linked list needs positional traversal.

C - Incorrect: A queue exposes front/rear service, not direct seat lookup.

D - Incorrect: A stack exposes its top rather than arbitrary seat positions.

Primary note proof

Study-note section DSA02

For equal-size elements, address(A[i]) = base + i × element_size under zero-based indexing. This arithmetic gives constant-time positional access in the usual RAM cost model. A row-major rectangular matrix uses base + (row × columns + column) × element_size.

Study-note section DSA08

A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 41-48

Application of Queue
Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
• Print server- maintains a queue of print jobs
• Disk Driver- maintains a queue of disk input/output requests
• Task scheduler in multiprocessing system- maintains priority queues of processes
• Telephone calls in a busy environment -maintains a queue of telephone calls
• Simulation of waiting line- models real life queues (e.g. supermarkets checkout, customer service line (Bank, custom, airport etc))

Question sources

National Exit Exam practice questions · Question 5

Computer Science Exit Exam practice collection · Question 129

Question 99 | Q01005

Which notation writes binary arithmetic operators between their operands and uses precedence and parentheses, as in ordinary written arithmetic?

A. Infix notation

B. Prefix notation

C. Postfix notation

D. None of the above

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. Infix notation

Infix notation places each binary operator between its two operands, using precedence rules and parentheses to establish grouping. Prefix puts the operator before its operands, and postfix puts it after them. The criterion asks for that objective familiar arithmetic layout rather than a universal claim about personal comfort. Since A names the between-operands form, the none-of-the-above alternative does not apply.

Why each choice is right or wrong

A - Correct: Infix places the operator between operands.

B - Incorrect: Prefix puts the operator before its operands.

C - Incorrect: Postfix puts the operator after its operands.

D - Incorrect: Infix satisfies the specified layout.

Primary note proof

Study-note section DSA13

> Infix places an operator between operands and uses precedence and parentheses. Prefix places it first; postfix places it last. Their fixed operator arity determines grouping. To convert infix to postfix, output operands immediately and hold operators on a stack. Pop higher-precedence operators before pushing a new operator; also pop equal precedence for left-associative operators. Parentheses temporarily override this order.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

> • Reversing Data and Reverse a List
> • Convert Decimal to Binary
> • Parsing - It is a logic that breaks into independent pieces for further processing
> • Backtracking
> Examples:
> 1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
> 2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Question sources

National Exit Exam practice questions · Question 19

Computer Science Exit Exam practice collection · Question 130

Question 100 | Q01181

Which option is outside the three standard named depth-first visitation orders for binary trees in this course? This asks about those standard order names, not whether other traversal algorithms can exist.

A. Randomized traversal

B. Preorder traversal

C. Postorder traversal

D. Inorder traversal

Hint

Compare the required access order and representation constraints with the defining operations of each option.

Answer A: A. Randomized traversal

The three standard named orders are preorder, inorder and postorder, determined respectively by visiting the node before, between or after its two child subtrees. Randomized visitation can be implemented, but it is not one of these three conventional names. Thus Randomized traversal is the correct exclusion under the explicit classification. The question no longer claims that such a traversal is impossible or never used.

Why each choice is right or wrong

A - Correct: Randomized visitation is possible but is not one of the three standard named orders being classified.

B - Incorrect: Preorder is one of the standard orders.

C - Incorrect: Postorder is one of the standard orders.

D - Incorrect: Inorder is one of the standard orders.

Primary note proof

Study-note section DSA20

> Preorder visits node,left,right; inorder left,node,right; postorder left,right,node. Apply recursively; null children do nothing. Inorder returns sorted keys only for a BST. Preorder supports copying, postorder supports deleting children before parents, and level order uses a FIFO queue.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 40-44

> III. Tree Traversing:
> Binary search tree can be traversed in three ways.
> a. Pre order traversal - traversing binary tree in the order of parent, left and right.

b. Inorder traversal - traversing binary tree in the order of left, parent and right.
c. Postorder traversal - traversing binary tree in the order of left, right and parent.

Question sources

Computer Science model exam 2 · Question 2

Question 101 | Q01182

Which of the following principles does Queue use?

A. LIFO principle

B. FIFO principle

C. Linear tree

D. Ordered array

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: FIFO principle

Insertion at rear and deletion at front preserve arrival order, which the source calls FIFO. Evidence: assigned PDF pages 7.

Why each choice is right or wrong

A - Incorrect: LIFO is the stack policy. See assigned PDF 7.

B - Correct: FIFO is explicitly named for queues. See assigned PDF 7.

C - Incorrect: Linear tree is not an ordering policy. See assigned PDF 7.

D - Incorrect: Ordered array describes neither the queue service rule nor all its representations. See assigned PDF 7.

Primary note proof

Study-note section DSA14

An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.

The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Computer Science model exam 2 · Question 3

Question 102 | Q01198

In this course's standard ADT vocabulary, which structure calls insertion at its top push and removal from that same top pop?

A. Queue

B. Stack

C. Both

D. None

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: B. Stack

The stack contract inserts and removes at one top. Its standard operations are push for insertion and pop for removal, with peek observing the top without removing it. The ordinary queue contract instead enqueues at the rear and dequeues at the front. The question fixes this conceptual vocabulary rather than comparing method names chosen by particular programming libraries, making Stack the definite selection.

Why each choice is right or wrong

A - Incorrect: The standard queue vocabulary is enqueue/dequeue at different endpoints.

B - Correct: Push and pop name the standard stack-top operations.

C - Incorrect: Both would incorrectly include the queue under the specified vocabulary.

D - Incorrect: Stack is present and matches the stated operations.

Primary note proof

Study-note section DSA11

Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA14

An ordinary queue enqueues at rear, dequeues at front, and peeks at front.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Computer Science model exam 2 · Question 19

Question 103 | Q01204

Which task requires FIFO service rather than the LIFO service of a stack used as the sole pending-item ADT? In D, asynchronous transfer must deliver buffered items in arrival order.

A. String reversal

B. Recursion

C. Backtracking

D. Asynchronous buffered transfer with mandatory arrival-order delivery

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: D. Asynchronous buffered transfer with mandatory arrival-order delivery

Stacks support reversal by popping items in reverse insertion order. They also manage nested recursive calls and backtracking through the most recent pending context or choice. The stated asynchronous-transfer contract instead requires older buffered items to be delivered before newer arrivals. That is FIFO service, so an ordinary queue supplies the needed ordering. A single stack's LIFO removal rule does not meet this explicitly ordered buffering contract.

Why each choice is right or wrong

A - Incorrect: Reversal uses the stack's reverse insertion order.

B - Incorrect: Nested recursive calls return most-recent-first.

C - Incorrect: Backtracking resumes the latest pending choice.

D - Correct: Arrival-order delivery requires FIFO rather than a single stack's LIFO.

Primary note proof

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Study-note section DSA14

> FIFO suits equal-priority printer jobs, shared-resource waiting, asynchronous buffers and level-order exploration. A play sequence can also use a queue. Distinguish airline seat records from waiting requests; an application name alone does not force one representation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

> • Reversing Data and Reverse a List
> • Convert Decimal to Binary
> • Parsing - It is a logic that breaks into independent pieces for further processing
> • Backtracking
> Examples:
> 1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
> 2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 6 | lines 46-52

> Applications of Stack
> • It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
> • Infix to Postfix Transformation
> • It is useful during the execution of recursive programs
> • A Stack is useful for designing the compiler in operating system to store local

variables inside a function block.
• A stack (memory stack) can be used in function calls including recursion.

Question sources

Computer Science model exam 2 · Question 25

Question 104 | Q01228

For an unordered symbol table with no range queries, assume suitable hashing and bounded load factor. Which listed ordinary representation offers expected constant-time lookup, compared with linear list search and logarithmic balanced-tree search?

A. Self-organizing list

B. Hash table

C. Balanced search tree

D. Linear list scan

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: B. Hash table

Symbol tables can use hashing for expected constant lookup when hashing spreads keys suitably and load remains bounded. Balanced search trees supply logarithmic lookup and ordering, while list search can require a linear scan. The comparison concerns the expected lookup growth under those assumptions, not an unconditional worst-case minimum. Collision-heavy hash inputs can still take linear time outside that expected analysis.

Why each choice is right or wrong

A - Incorrect: Ordinary self-organizing list search does not guarantee the stated general expected bound.

B - Correct: Suitable hashing with bounded load provides expected constant lookup.

C - Incorrect: A balanced search tree has logarithmic lookup.

D - Incorrect: A linear list scan takes linear worst-case lookup.

Primary note proof

Study-note section DSA27

A dictionary or compiler symbol table can use hashing for expected constant lookup, or a balanced tree for logarithmic lookup with ordering. A bounded cache also needs eviction rules: FIFO uses a queue; LRU can combine hash lookup with a doubly linked recency list. Move a hit to the recent end and evict the least-recent end when full. No storage structure is universally best without operation and policy requirements.

Study-note section DSA26

Separate chaining stores colliding entries in a bucket list. Load alpha=n/m is average chain length; expected operations are O(1+alpha) under suitable distribution assumptions, with O(n) worst cases. Open addressing keeps entries inside the array and probes until a key or an available slot is found.

Question sources

Computer Science model exam 2 · Question 49

Question 105 | Q01241

In the linked list implementation of a queue, where will the new element be inserted?

A. At the middle position of the linked list.

B. At the head position of the linked list.

C. At the tail position of the linked list.

D. None of the above.

Hint

Trace the initial state and update only the fields affected by each permitted operation.

Answer C: At the tail position of the linked list.

The linked queue implementation expressly inserts a node at the end and removes the first node. End means tail. Evidence: assigned PDF pages 8.

Why each choice is right or wrong

A - Incorrect: Middle insertion would break the specified FIFO insertion rule. See assigned PDF 8.

B - Incorrect: Head insertion is the removal end in this representation. See assigned PDF 8.

C - Correct: New nodes are inserted at the linked-list end/tail. See assigned PDF 8.

D - Incorrect: C matches the implementation. See assigned PDF 8.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 35-37

> III. Linked list implementation of enqueue and dequeue operations
> Enqueue- is inserting a node at the end of a linked list
> Dequeue- is deleting the first node in the list

Question sources

Computer Science model exam 2 · Question 62

Question 106 | Q01263

What is the maximum number of children that a node can have in a binary tree?

A. 3

B. 1

C. 4

D. 2

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: 2

A binary tree has at most two children per node, named left and right. Maximum is therefore two. Evidence: assigned PDF pages 9.

Why each choice is right or wrong

A - Incorrect: Three exceeds the binary limit. See assigned PDF 9.

B - Incorrect: One is permitted but is not the maximum. See assigned PDF 9.

C - Incorrect: Four exceeds the binary limit. See assigned PDF 9.

D - Correct: Two equals the specified maximum. See assigned PDF 9.

Primary note proof

Study-note section DSA18

> Binary means at most two children. Full means each node has zero or two children. Complete means all earlier levels are filled and the last fills from left. Perfect means all leaves share a level and every internal node has two children. Balanced means height stays logarithmic under a stated rule; it does not require a perfect shape.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 14-19

> • Ancestors of a node: parent, grandparent, grand-grandparent, etc of a node.
> • Descendants of a node: children, grandchildren, grand-grandchildren etc of a node.
> • Depth of a node: number of ancestors or length of the path from the root to the node.
> • Height of a tree: depth of the deepest node.
> • Subtree: a tree consisting of a node and its descendants.
> • Binary tree: a tree in which each node has at most two children called left child and right child.

Question sources

Computer Science model exam 2 · Question 84

Question 107 | Q01442

DAG is an abbreviation of?

A. Detecting Acyclic Graph

B. Data Acyclic Graph

C. Dynamic Acyclic Graph

D. Directed Acyclic Graph

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: Directed Acyclic Graph

DSA40 directly expands DAG to directed acyclic graph. Directed specifies that each edge has an orientation, and acyclic excludes directed cycles. Detecting, Data, and Dynamic do not supply the defined first word. This is a vocabulary question and does not depend on graph representation or a particular traversal algorithm.

Why each choice is right or wrong

A - Incorrect: Detecting is not the first word in the defined expansion of DAG.

B - Incorrect: Data is not the first word in the defined graph-theoretic abbreviation DAG.

C - Incorrect: Dynamic is not the first word of DAG as defined in DSA40.

D - Correct: Directed Acyclic Graph is the exact expansion explicitly given in the complete note.

Primary note proof

Study-note section DSA40

> A graph has vertices V and edges E. Directed edges distinguish u→v from v→u; undirected edges represent two-way relationships. Following accounts is directed; a mutual friendship relation is undirected. Weighted edges carry costs; unweighted shortest-path questions count edges. DAG expands to directed acyclic graph. Trees are connected acyclic graphs, although graph data structures may also contain cycles.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 15-18

> a. Linear data structure: data items stores in sequence
> Example: Array, stack and Queue,
> b. Non-Linear data structures : in which there is order of data items
> Example: Trees and Graphs

Question sources

Compiler Design review questions · Question 50

Question 108 | Q01497

A binary search tree was built by inserting the following numbers in the given order: 40, 20, 50, 10, 30, 60. Which of the following statements are true?

A. 30 is the left child of 40

B. the tree is a heap

C. height of the tree is 3

D. 50 is the right child of 40

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: 50 is the right child of 40

Insertions give root 40; its children 20 and 50; 20 has children 10 and 30; 50 has right child 60. The deepest depth is two edges. A max heap is impossible because child 50 exceeds root 40. Evidence: assigned PDF pages 9, 11.

Why each choice is right or wrong

A - Incorrect: 30 is the right child of 20, not the left child of 40. See assigned PDF 9, 9, 9, 11.

B - Incorrect: 50 exceeds its parent 40, violating the module max-heap rule. See assigned PDF 9, 9, 9, 11.

C - Incorrect: Height is depth two, not the three-level count. See assigned PDF 9, 9, 9, 11.

D - Correct: 50 is larger than 40 and becomes its right child. See assigned PDF 9, 9, 9, 11.

Primary note proof

Study-note section DSA19

> With distinct keys, every key in a node’s left subtree is smaller and every key in its right subtree is larger. Immediate-child comparisons alone are insufficient: root 10, left child 5, and 5’s right child 12 violate the root’s restriction. Duplicates require a separate count or placement policy.

Study-note section DSA18

> A rooted tree has one parentless root and one parent per other node. It is connected and acyclic, has n-1 edges, and one path between any pair. Depth counts root-to-node edges; height is maximum depth. A single-node tree therefore has height zero. Leaves have no children.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

> • Binary tree: a tree in which each node has at most two children called left child and right child.
> • Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
> satisfies the following.

o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Data Structure and Algorithms module | PDF page 9 | lines 33-39

II. Insertion
When a node is inserted the definition of binary search tree should be preserved.
Case 1: There is no data in the tree (i.e. RootNodePtr is NULL)
• The node pointed by InsNodePtr should be made the root node.
Case 2: There is data
• Search the appropriate position.
• Insert the node in that position.

Data Structure and Algorithms module | PDF page 9 | lines 14-19

• Ancestors of a node: parent, grandparent, grand-grandparent, etc of a node.
• Descendants of a node: children, grandchildren, grand-grandchildren etc of a node.
• Depth of a node: number of ancestors or length of the path from the root to the node.
• Height of a tree: depth of the deepest node.
• Subtree: a tree consisting of a node and its descendants.
• Binary tree: a tree in which each node has at most two children called left child and right child.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Mock exam 1 · Question 13

Question 109 | Q01498

Certain operations on lists can be performed faster when the list is sorted than not. Which of the following could be an example of such an operation?

A. Searching for an element of a list

B. Computing the variance of a list

C. Computing the average of a list

D. All can be performed faster when the list is sorted.

E. Counting for the number of odd numbers in a list

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: Searching for an element of a list

Sortedness permits logarithmic binary search instead of a linear scan. Sorting alone does not remove the need to inspect all values for mean, variance or arbitrary parity counts. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Correct: Binary searching uses sorted order to discard half the list at each step. See assigned PDF 3, 3.

B - Incorrect: Variance still depends on all values, without extra aggregates. See assigned PDF 3, 3.

C - Incorrect: Average still depends on all values, without a stored sum. See assigned PDF 3, 3.

D - Incorrect: The statistical computations are not accelerated solely by ordering. See assigned PDF 3, 3.

E - Incorrect: Odd and even values are not a monotone split in numeric sorted order. See assigned PDF 3, 3.

Primary note proof

Study-note section DSA28

> For a single unsorted query, sorting first is rarely justified. Small lists may favour scanning because setup matters. Sorting does not avoid examining all values for mean, variance or arbitrary parity counts.

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Mock exam 1 · Question 14

Question 110 | Q01499

For the values 5 3 8 9 1 7 0 2 6 4, which offered arrays satisfy the final-pivot partition condition for pivot 5: all smaller values precede it and all larger values follow it? Do not require a particular partition algorithm or internal ordering. Select all that apply.

A. 3 1 0 2 4 5 8 9 7 6

B. None of the choices are correct

C. 4 2 3 0 1 5 7 9 6 8

D. 0 3 4 2 1 5 7 9 6 8

E. 3 1 0 2 4 5 8 9 6 7

F. 5 3 4 2 1 0 7 9 6 8

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A, C, D, E: A. 3 1 0 2 4 5 8 9 7 6; C. 4 2 3 0 1 5 7 9 6 8; D. 0 3 4 2 1 5 7 9 6 8; E. 3 1 0 2 4 5 8 9 6 7

There are five values below 5 and four values above it, so a final-position pivot must occupy index five in zero-based order. A, C, D and E each place precisely the smaller group before that pivot and the larger group after it. Their internal subgroup orders need not agree because only the partition invariant is requested. F leaves smaller values after an initial pivot and fails the condition.

Why each choice is right or wrong

A - Correct: All five smaller values precede 5 and all larger ones follow.

B - Incorrect: Several offered arrays satisfy the condition.

C - Correct: This array satisfies the smaller/pivot/larger grouping.

D - Correct: This array also satisfies the grouping despite different internal order.

E - Correct: The right-side permutation does not violate the grouping.

F - Incorrect: Smaller values after the leading 5 violate its final-position condition.

Primary note proof

Study-note section DSA36

> In the ten-distinct-value example, a final pivot 5 has five smaller values before it and four larger after it. Internal orders depend on the partition algorithm; several candidate arrays can satisfy the invariant. State the algorithm before predicting one exact array. Quicksort normally rearranges in the same array but recursive storage can be linear in the worst case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

> ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
> Algorithm:
> 1. Choose a pivot value (mostly the first element is taken as the pivot value)
> 2. Position the pivot element and partition the list so that:
> • the left part has items less than or equal to the pivot value
> • the right part has items greater than or equal to the pivot value
> 3. Recursively sort the left part
> 4. Recursively sort the right part

Question sources

Mock exam 1 · Question 15

Question 111 | Q01578

An owning dynamically allocated linked-list API must provide item search, insertion and removal. Removal unlinks and frees its node before returning; it returns any needed payload by value. Which additional function is unnecessary and would violate the contract if its result were dereferenced?

A. A function that returns a pointer to an already freed node for later dereferencing

B. A function that searches for an item from the list

C. A function that inserts an item in the list

D. A function that removes an item from the list

E. None of the choices is correct

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: A. A function that returns a pointer to an already freed node for later dereferencing

The API explicitly requires search, insertion and removal, so those operations belong in its interface. Its removal contract frees the node before returning, leaving that node's storage unavailable for later dereference. An additional function promising a usable pointer to the freed node is therefore inappropriate. Returning a copied payload is different and remains allowed; the distinction concerns storage lifetime rather than whether deletion may report a value.

Why each choice is right or wrong

A - Correct: A usable pointer to a freed node violates the stated lifetime contract.

B - Incorrect: Item search is explicitly required by this API.

C - Incorrect: Item insertion is explicitly required by this API.

D - Incorrect: Item removal is explicitly required by this API.

E - Incorrect: A identifies an inappropriate extra function.

Primary note proof

Study-note section DSA03

> A C struct groups named fields, including arrays and pointers. A singly linked node needs payload plus a same-type link. C fields cannot have initializers inside the type declaration; modern C++ permits default member initializers. In C++, match new with delete and new[] with delete[]. Repair links before freeing a node; never dereference it afterwards.

Study-note section DSA09

> Prepend with new.next=head; head=new. To insert after a known predecessor p, set new.next=p.next before p.next=new. Appending with only head requires a tail traversal; maintaining tail avoids it. Delete the head by saving old=head, advancing head=head.next and releasing old in C++.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
> to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 35-41

> IV. Deleting a node from the list
> When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the delete instruction:
> delete temp; // Release the memory pointed to by temp
> However, we can't just delete the nodes willy-nilly as it would break the chain. We need
> to reassign the pointers and then delete the node at the last moment. We can delete nodes
> at the start, middle or end of the list. Here let’s see the steps to delete at the end.

Question sources

Mock exam 3 · Question 10

Question 112 | Q01579

Consider a max-heap of n>=3 distinct keys, stored in ordinary level-order array form. Its binary tree is full: every internal node has two children. Which statement is false?

A. The root is the biggest element.

B. The second biggest element is necessarily found at depth 1 (root is at depth 0)

C. The search for the smallest element can be accomplished in O(log n) time (worst case)

D. The last element of the array is necessarily the right most child on the lowest level.

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: C is false: finding the minimum can require Theta(n) time.

The root contains the maximum key. With distinct keys, the second largest must be one of its children, since a deeper key has a larger ancestor above it. Completeness places the final occupied array slot at the rightmost occupied position of the lowest level. The minimum can occur at any leaf, however, so finding it may require a linear scan rather than logarithmic worst-case time. C is therefore false.

Why each choice is right or wrong

A - Incorrect: The max-heap order makes the root the largest key, so this is true rather than the requested false statement.

B - Incorrect: Distinct keys ensure the second largest is a root child, so this is true.

C - Correct: The minimum may require scanning linearly many leaves; the logarithmic worst-case claim is false.

D - Incorrect: Level-order storage of a complete heap ends at the rightmost occupied position on the lowest level.

Primary note proof

Study-note section DSA23

> A binary max-heap is complete with each parent at least its children; a min-heap reverses this order. Unlike BSTs, heaps do not order entire subtrees. The root is maximum. If all keys are distinct, the second largest is a root child. With duplicate maxima, the second largest distinct value may be deeper: [9,9,9,8,7,6,5] is a valid max-heap and that value is 8 at index 3. The minimum may be any leaf, requiring linear worst-case scanning.

Study-note section DSA18

> Complete means all earlier levels are filled and the last fills from left.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

> iii. Heap Sort
> Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
> each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
> Algorithm:
> 1. Construct a binary tree
> • The root node corresponds to Data[0].
> • If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.

2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Data Structure and Algorithms module | PDF page 9 | lines 14-19

- Ancestors of a node: parent, grandparent, grand-grandparent, etc of a node.
- Descendants of a node: children, grandchildren, grand-grandchildren etc of a node.
- Depth of a node: number of ancestors or length of the path from the root to the node.
- Height of a tree: depth of the deepest node.
- Subtree: a tree consisting of a node and its descendants.
- Binary tree: a tree in which each node has at most two children called left child and right child.

Question sources

Mock exam 3 · Question 11

Question 113 | Q01580

For ordinary top-down array mergesort on n=8 items with left-first ties and no merge-skipping, which description alone guarantees the maximum possible number of key comparisons across the merges?

A. When the list is initially sorted in the "reverse" order

B. When the list is already sorted in the "right" order

C. When the list contains all similar elements

D. When the list contains all distinct elements

E. None of the choices is correct

Hint

Compare the required operations, access order and representation metadata with each option.

Answer E: E. None of the choices is correct

Maximum merge comparisons depend on the interleaving of the sorted runs, rather than merely on a broad input label. Sorted or reverse-sorted input produces separated runs, and equal keys let left-first merging exhaust a run early. Distinct keys permit a worst interleaving but do not force one; already sorted distinct keys are a counterexample. None of A-D therefore guarantees the requested exact maximum.

Why each choice is right or wrong

A - Incorrect: Reverse order alone does not force worst interleavings.

B - Incorrect: Sorted runs can be merged with an early exhaustion of one side.

C - Incorrect: Left-first equal keys exhaust a run without maximum interleaving.

D - Incorrect: Distinct keys can already be sorted, so distinctness is insufficient.

E - Correct: No preceding description alone guarantees the exact maximum comparison count.

Primary note proof

Study-note section DSA37

> Split into halves, sort them, then merge by advancing the smaller current front. With left-first tie handling, equal keys retain order. Ordinary array merging needs a linear auxiliary buffer. There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA37

> Reverse order alone does not characterize maximum merge comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

> iv. Merge sort
> Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
> Algorithm:
> 1. Divide the array in to two halves.
> 2. Recursively sort the first n/2 items.
> 3. Recursively sort the last n/2 items.
> 4. Merge sorted items (using an auxiliary array).

Question sources

Mock exam 3 · Question 12

Question 114 | Q01618

If a sorted list contains 100 elements, what is the maximum number of comparisons to make to search for an element using binary searching algorithm?

A. None of the choices are correct

B. 100

C. 10

D. 7

E. 50

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: 7

With one midpoint-key comparison per visited node, the maximum probes are floor(log2 100)+1=7, since 64<=100<128. Equivalently seven binary levels can distinguish up to 127 keys. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: Seven is present as D. See assigned PDF 3.

B - Incorrect: 100 is a complete linear scan. See assigned PDF 3.

C - Incorrect: 10 exceeds the needed halving depth. See assigned PDF 3.

D - Correct: Seven midpoint probes suffice in the worst case. See assigned PDF 3.

E - Incorrect: 50 counts only the first halving, not the number of probes. See assigned PDF 3.

Primary note proof

Study-note section DSA29

> Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

Mock exam 2 · Question 7

Question 115 | Q01619

For ordinary representations without an extra positional index, which listed structure guarantees O(1) worst-case access to an arbitrary valid logical position as its length grows? Use fixed-size elements and unit-cost address arithmetic.

A. Singly linked list

B. Doubly linked list

C. Every listed linked structure guarantees constant-time access to an arbitrary logical position.

D. Circular linked list

E. Array

Hint

Compare the required operations, access order and representation metadata with each option.

Answer E: E. Array

An array computes an element address from its base address, the requested index and the fixed element size, giving constant-time access. Ordinary singly, doubly and circular linked lists instead follow links to reach a logical position; even starting from the nearer endpoint leaves linear worst-case traversal. The replacement aggregate claim is false because those linked structures lack a direct positional index. Array is therefore the unique matching choice.

Why each choice is right or wrong

A - Incorrect: A singly linked list may traverse linearly many successor links.

B - Incorrect: A doubly linked list still has linear worst-case access to middle positions.

C - Incorrect: The listed linked structures do not all have constant-time arbitrary positional lookup.

D - Incorrect: Circularity changes how a chain ends, not how a logical position is located.

E - Correct: An array computes the address directly with fixed-size element arithmetic.

Primary note proof

Study-note section DSA02

> For equal-size elements, address(A[i]) = base + i × element_size under zero-based indexing. This arithmetic gives constant-time positional access in the usual RAM cost model.

Study-note section DSA08

> An array-linked list still follows links unless separately position-indexed.

Study-note section DSA10

> Finding a position is still linear.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 26-34

III. Displaying the list of Nodes
After adding nodes to the list, we need to display the list of nodes on the screen. Here are the steps required to display all nodes item in the list:
1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the next pointer of the node it is currently indicating.
5. Jump back to step 2.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

I. Array implementation (static memory).
Push(n): Adding item at top
top is the current top of stack and n is its
maximum size
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;

Question sources

Mock exam 2 · Question 8

Question 116 | Q01620

Which listed representations guarantee O(log n) worst-case arbitrary-key lookup under these implementations? Use a height-balanced BST for A, an ordinary singly linked list for C, an ordinary binary heap for D, and a sorted random-access array with binary search for E. Select all that apply.

A. Height-balanced BST

B. A and C

C. Singly linked list

D. Heap

E. Sorted random-access array with binary search

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A, E: A. Height-balanced BST; E. Sorted random-access array with binary search

A height-balanced BST has logarithmic height, so following its ordered search path takes logarithmic worst-case time. A sorted random-access array supports logarithmic binary-search probes. An ordinary singly linked list must traverse links, and a heap does not order entire subtrees by arbitrary search keys. Thus the combined A-and-C claim fails because C lacks the guarantee, while A and E independently meet it.

Why each choice is right or wrong

A - Correct: Balance bounds the BST search path logarithmically.

B - Incorrect: C lacks the logarithmic guarantee, so A and C is false.

C - Incorrect: Singly linked lookup may scan all nodes.

D - Incorrect: Heap order privileges the extreme but not arbitrary-key lookup.

E - Correct: Binary search on a sorted random-access array is logarithmic.

Primary note proof

Study-note section DSA19

> Search compares at each node and follows one child until equality or null. Insertion uses that same path to attach a new leaf. Operations cost O(height): logarithmic for balanced trees, linear for a chain. Best-case successful search is Theta(1), a root hit. An optimal BST minimizes expected search cost using access probabilities; constructing it is a different problem from searching it.

Study-note section DSA29

Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Study-note section DSA23

Unlike BSTs, heaps do not order entire subtrees.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates
mid-point and identify which have the target
belongs to since the list is sorted. It performs
this recursively till the item found or the list
is empty(target is not in the list
• The computational time for this algorithm is
proportional to log2 n.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Mock exam 2 · Question 9

Question 117 | Q00426B

A queue is a ?

A. FIFO (First In First Out) list

B. LIFO (Last In First Out) list

C. Ordered array

D. Linear tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: FIFO (First In First Out) list

The source expressly calls queues First in First out lists. Evidence: assigned PDF pages 7.

Why each choice is right or wrong

A - Correct: FIFO preserves arrival order. See assigned PDF 7.

B - Incorrect: LIFO names a stack. See assigned PDF 7.

C - Incorrect: An ordered array is one possible storage form, not the abstract queue rule. See assigned PDF 7.

D - Incorrect: Linear tree is not the source queue definition. See assigned PDF 7.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Data Structures & Algorithms practice questions · Question 56

Question 118 | Q00394

What is an internal sorting algorithm?

A. Algorithm that uses tape or disk during the sort

B. Algorithm that uses main memory during the sort

C. Algorithm that involves swapping

D. Algorithm that are considered 'in place'

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: Sorting whose working dataset fits in main memory.

DSA37 defines internal sorting by fitting the working dataset in memory. Option B expresses that main-memory setting. In contrast, external sorting handles disk-backed runs when data exceeds memory. Swapping is an operation used by some algorithms, and in-place describes auxiliary storage; neither determines whether a sort is internal.

Why each choice is right or wrong

A - Incorrect: Tape or disk working data characterizes external sorting rather than the main-memory internal setting.

B - Correct: Internal sorting keeps its working dataset in main memory, matching the definition in DSA37.

C - Incorrect: Swapping is an implementation operation and does not define the location of the working dataset.

D - Incorrect: In-place concerns auxiliary element storage; an internal sort may still use a separate memory buffer.

Primary note proof

Study-note section DSA37

> Top-down mergesort recurses; bottom-up merges runs of widths 1,2,4,... iteratively. In-place merging variants exist with different tradeoffs; linear merge names an operation, not a universally standardized sort variant. Internal sorting fits the working dataset in memory. External sorting builds sorted runs and merges buffered streams from disk, aiming to reduce I/O. Reverse order alone does not characterize maximum merge comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 33-37

> Algorithm analysis refers to the process of determining the amount of computing time and storage
> space required by different algorithms. It used to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexityof the

algorithm.
Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

Question sources

Data Structures & Algorithms practice questions · Question 24

Question 119 | SM0008

What is the time complexity order of binary searching algorithm?

A. O(n^3)

B. O(n)

C. O(log n)

D. O(1)

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: O(log n)

The stated time is proportional to log2 n because each midpoint decision discards half the remaining ordered interval. Evidence: assigned PDF pages 3.

Why each choice is right or wrong

A - Incorrect: Cubic does not match repeated halving. See assigned PDF 3.

B - Incorrect: Linear is the source sequential-search order. See assigned PDF 3.

C - Correct: Logarithmic is the explicit binary-search order. See assigned PDF 3.

D - Incorrect: Constant time occurs for an immediate match, not a general search bound. See assigned PDF 3.

Primary note proof

Study-note section DSA29

Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list

• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

MOE Computer Science Exit Exam · Question 2-4

National Computer Science Exit Exam · Question 62

National Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 108

Question 120 | SM0016

What is the tightest listed worst-case time bound for ordinary quicksort when each partition may choose an extreme pivot?

A. O(n)

B. O(n^2)

C. O(2^n)

D. O(n log n)

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: B. O(n^2)

An extreme pivot leaves one empty part and one part with only a single element removed. Repeating that pattern makes partition work proportional to n+(n-1)+... rather than distributing it across balanced levels. The sum is quadratic. Balanced partitions explain the n log n alternative, but they are not guaranteed in this worst case. Exponential time is a looser upper bound, not the tightest listed one.

Why each choice is right or wrong

A - Incorrect: Linear time does not cover repeated partition scans.

B - Correct: Repeated extreme pivots produce quadratic total work.

C - Incorrect: Exponential is a much looser upper bound.

D - Incorrect: n log n requires suitably balanced or expected partitions.

Primary note proof

Study-note section DSA36

Choose a pivot, partition, then recursively sort the pieces. Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work. Expected n log n needs an input/random-pivot model. Random and median-of-three choices have different tradeoffs; neither is universally best.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Question sources

MOE Computer Science Exit Exam · Question 4-4

National Computer Science Exit Exam · Question 74

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 109

Question 121 | SM0043

Under universal hashing, one random family member is chosen for a table independently of a fixed key set. Which claims are false? Interpret C as an unconditional superiority claim. Select all that apply.

A. In universal hashing a hash function is selected randomly from a set of hash functions for each key

B. In universal hashing a hash function is selected from a set of hash functions independent of keys

C. A single fixed hash function is always better than universal hashing at reducing collisions.

D. Universal hashing controls collision probability without guaranteeing zero collisions.

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A, C: A. In universal hashing a hash function is selected randomly from a set of hash functions for each key; C. A single fixed hash function is always better than universal hashing at reducing collisions.

Universal hashing chooses one random function for the table and then uses it consistently, rather than selecting an unrelated function afresh for each key. Its purpose is to control collision probability for a fixed key set. It does not establish that a single fixed function is always better for avoiding collisions. Thus A and C are false, while independent table-level choice and probability control are the valid principles.

Why each choice is right or wrong

A - Correct: The chosen function is reused for the table rather than chosen afresh per key.

B - Incorrect: Independence from the fixed key set is the stated selection principle.

C - Correct: Unconditional fixed-function superiority does not follow and contradicts the randomized collision-control purpose.

D - Incorrect: Universal choice controls collision probability without promising no collisions.

Primary note proof

Study-note section DSA25

> Determinism means repeated equal keys get the same address for the chosen function. Uniformity means expected keys spread evenly. Universal hashing chooses one function randomly from a suitable family for the table, independently of a fixed key set; it does not choose a fresh unrelated function for each lookup. Random choice controls collision probability, not a promise of zero collisions.

Question sources

MOE Computer Science Exit Exam · Question 13-1

National Computer Science Exit Exam · Question 25

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 124

Question 122 | SM0050

In a tree data structure, if a node has no parent node, then the node is _____.

A. Parent node

B. Root node

C. External node

D. Internal node

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: Root node

The module defines root as a node without a parent, whereas leaves have no child and internal nodes have children. Evidence: assigned PDF pages 9.

Why each choice is right or wrong

A - Incorrect: Parent describes a relation to a child, not the absence of a parent. See assigned PDF 9.

B - Correct: Root is explicitly the parentless node. See assigned PDF 9.

C - Incorrect: External nodes are defined by having no child. See assigned PDF 9.

D - Incorrect: Internal nodes have at least one child, which does not imply parentlessness. See assigned PDF 9.

Primary note proof

Study-note section DSA18

> A rooted tree has one parentless root and one parent per other node. It is connected and acyclic, has n-1 edges, and one path between any pair. Depth counts root-to-node edges; height is maximum depth. A single-node tree therefore has height zero. Leaves have no children.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 3-13

> A tree is a set of nodes and edges that connect pairs of nodes that connect pairs of nodes. It is an abstract model of a hierarchical structure. Rooted tree has the following structure:
> • One node distinguished as root.
> • Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
> • There is a unique path from the root to the each node.
> • The number of edges in a path is the length of the path.
> Tree Terminologies:
> • Root: a node without a parent.
> • Internal node: a node with at least one child.
> • External (leaf) node: a node without a child.

Question sources

MOE Computer Science Exit Exam · Question 15-1

National Computer Science Exit Exam · Question 7

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 123

Question 123 | SM0053

Linked list data structure is _____ type of data structure.

A. Non-linear data structure

B. Fixed size data structure

C. Static data structure

D. Dynamic data structure

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: Dynamic data structure

Linked nodes are created using new and the module labels linked-list memory dynamic. The size is not fixed by a static array declaration. Evidence: assigned PDF pages 5, 6.

Why each choice is right or wrong

A - Incorrect: The supplied single linked list forms a linear chain. See assigned PDF 5, 6.

B - Incorrect: New nodes can be added at runtime, so size is not intrinsically fixed. See assigned PDF 5, 6.

C - Incorrect: Static describes the source array implementation. See assigned PDF 5, 6.

D - Correct: Dynamic allocation is the linked representation given. See assigned PDF 5, 6.

Primary note proof

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 5 | lines 3-8

> While adding a node to a list, the chain needs to be rearranged properly so that the each nodes are connected to each other. It is possible to add a node anywhere in the chain (at the start, middle or end). Let’s consider adding a node at the end of the list:
> Step 1: First we need to reserve a space for the new node using the keyword new.
> Node *temp = new *Node; // Reserving a space for temporary pointer which points to new node;

Data Structure and Algorithms module | PDF page 6 | lines 43-45

II. Linked list (dynamic memory)
The linked list implementation of stack is adding items at the end of the chain (Push item) and deleting the end node of the list (Pop item).

Question sources

MOE Computer Science Exit Exam · Question 15-4

National Computer Science Exit Exam · Question 95

National Computer Science Exit Exam

MOE Computer Science Exit Exam

MOE Computer Science Exit Exam · Question 27-4

National Computer Science Exit Exam · Question 99

Computer Science Exit Exam practice collection · Question 128

Question 124 | SM0064

Assume there is a line of patients waiting for service from a clinic. Seriously sick patients should get service first. Which data structure is the most appropriate for this real-world problem?

A. Tree

B. Queue

C. Stack

D. Priority Queue

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: Priority Queue

Give each patient a key representing medical urgency and serve by that priority. This adapts the module associated-key priority queue to the stated urgency rule. Evidence: assigned PDF pages 8, 7.

Why each choice is right or wrong

A - Incorrect: A tree alone does not define urgency-based service. See assigned PDF 8, 7.

B - Incorrect: A plain FIFO queue serves arrival order rather than illness severity. See assigned PDF 8, 7.

C - Incorrect: A stack gives recent arrivals precedence, regardless of severity. See assigned PDF 8, 7.

D - Correct: Associated priority keys represent how urgently each patient needs service. See assigned PDF 8, 7.

Primary note proof

Study-note section DSA17

A priority queue returns the smallest or largest priority key under its convention. Urgent patients may precede earlier arrivals. Insertions and extractions can interleave. Binary heaps support constant peek and logarithmic insert/extract.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 40-45

Priority Queue:- is a queue where each data has an associated key provided at insertion time.
Application of Queue
Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
• Print server- maintains a queue of print jobs
• Disk Driver- maintains a queue of disk input/output requests
• Task scheduler in multiprocessing system- maintains priority queues of processes

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

MOE Computer Science Exit Exam · Question 18-3

National Computer Science Exit Exam · Question 34

National Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 125

Question 125 | SM0076

For an input graph, Depth First Traversal (DFT) algorithm uses _____ data structure to traverse.

A. Priority queue

B. Tree

C. Queue

D. Stack

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: Stack

DSA42 says DFS recursively explores an unvisited neighbor to completion before returning. The recursive call stack stores return points; an explicit stack can store pending work instead. Both implement latest-pending-work-first order. A queue would change that discipline, while a priority queue or tree alone does not encode the required depth-first frontier behavior.

Why each choice is right or wrong

A - Incorrect: Priority service follows priority keys and does not guarantee finishing the latest active branch first.

B - Incorrect: A tree may represent traversal results, but it does not identify the standard DFS pending-work discipline.

C - Incorrect: FIFO queue order produces breadth-first pending work rather than the required latest-pending-work behavior.

D - Correct: A stack, explicit or supplied by recursion, preserves return points and latest-pending-work-first order.

Primary note proof

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

> • Reversing Data and Reverse a List
> • Convert Decimal to Binary
> • Parsing - It is a logic that breaks into independent pieces for further processing
> • Backtracking
> Examples:
> 1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
> 2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.

Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

MOE Computer Science Exit Exam · Question 22-1

National Computer Science Exit Exam · Question 27

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 126

Question 126 | SM0082

Which statement is true about the pop() operation of Stack data structures?

A. pop() returns the end/top element but it doesn't delete that element

B. pop() removes the end/top element from the stack

C. pop() inserts a new element at the middle of stack

D. pop() inserts a new element at the beginning/bottom of stack

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: pop() removes the end/top element from the stack

The pop pseudocode reads the current top and decrements top, so the element is logically removed. Merely inspecting without deletion is peek. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: This describes peek, not pop. See assigned PDF 6, 6.

B - Correct: Pop removes the current top by decreasing top. See assigned PDF 6, 6.

C - Incorrect: Pop does not insert, and stack insertion is only at top. See assigned PDF 6, 6.

D - Incorrect: Pop is removal, not bottom insertion. See assigned PDF 6, 6.

Primary note proof

Study-note section DSA11

Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

"A stack is an ordered list in which all insertions and deletions are made at one end, called the top". Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 6 | lines 24-42

I. Array implementation (static memory).
Push(n): Adding item at top
top is the current top of stack and n is its
maximum size
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;

Question sources

MOE Computer Science Exit Exam · Question 23-3

National Computer Science Exit Exam · Question 8

National Computer Science Exit Exam

MOE Computer Science Exit Exam

Computer Science Exit Exam practice collection · Question 127

Question 127 | SM0210

A singly linked list of n nodes maintains only its head; there is no tail pointer or index. Appending traverses to the last node before linking a new node. Which listed bounds hold for its worst-case time? Select all that apply.

A. O(n)

B. Theta(1)

C. Theta(n)

D. O(1)

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A, C: A. O(n); C. Theta(n)

The append operation begins at the head and must follow the chain to locate its final node. That requires linear traversal, followed by a constant amount of link work. Its tight worst-case bound is therefore Theta(n), which also implies the upper bound O(n). Neither constant-time choice holds as the list grows. Maintaining a separate tail would change the operation cost, but that representation is explicitly excluded.

Why each choice is right or wrong

A - Correct: O(n) is a valid upper bound on the traversal.

B - Incorrect: The growing traversal prevents a constant tight bound.

C - Correct: Theta(n) is the tight traversal bound.

D - Incorrect: O(1) does not bound arbitrary head-only append traversal.

Primary note proof

Study-note section DSA09

> Prepend with new.next=head; head=new. To insert after a known predecessor p, set new.next=p.next before p.next=new. Appending with only head requires a tail traversal; maintaining tail avoids it. Delete the head by saving old=head, advancing head=head.next and releasing old in C++.

Study-note section DSA06

> For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 5 | lines 13-24

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.

```
if (start == NULL) // If list is empty, then new node become start node
start = temp;
else {
temp2 = start; // list not empty! Thus temp2 starts from first node
while (temp2->next!= NULL) { // till the current last node found navigates forward
temp2 = temp2->next; // Move to next link in chain
}
temp2->next = temp; // new node (temp) appended at the end of the list
```

Data Structure and Algorithms module | PDF page 3 | lines 2-9

I. Big-Oh Notation (O): It's only concerned with what happens for very a large value of n. Big-O expresses an upper bound on the growth rate of a function, for sufficiently large values of n. An upper bound is the best algorithmic solution that has been found for a problem. NB: Each of the following functions is big-O of its successors:
II. Big-Omega Notation (Ω): provides an asymptotic lower bound; f(n)= Ω(g (n)) means that the growth rate of f(n) is greater than or equal to g(n).
III. Theta Notation (Θ): If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n) i.e f(n) and g(n) have the same rate of growth.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 3-3

Computer Science Exit Exam practice collection · Question 138

Question 128 | SM0213

Which ordinary search method requires its input key sequence to be sorted before using comparisons to discard the half that cannot contain a target?

A. Hash search

B. Depth-first search

C. Binary search

D. Linear search

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: C. Binary search

Binary search compares a target with the midpoint of an ordered interval, then safely discards one half. That elimination depends on sorted order. Linear search can scan unsorted values, hashing selects addresses from keys rather than sorted positions, and depth-first search follows graph adjacency. The revised alternatives therefore have clear operational meanings, and Binary search uniquely matches the stated ordered-half-elimination requirement.

Why each choice is right or wrong

A - Incorrect: Hash lookup uses computed addresses rather than a sorted input interval.

B - Incorrect: Depth-first search follows adjacency, not ordered-array halves.

C - Correct: Binary search needs ordering to safely discard a comparison-selected half.

D - Incorrect: Linear scanning permits unsorted input.

Primary note proof

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1.

Study-note section DSA28

> Unsorted data is allowed; sorted data is allowed too.

Study-note section DSA25

> A hash table uses h(key) in 0..m-1 and stores only actual entries plus bucket capacity.

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science Exit Exam · Sene 2016 · Question 4-3

Computer Science Exit Exam practice collection · Question 139

Question 129 | SM0229

Given a binary search tree, which traversal type would print the values in the nodes in sorted order?

A. Pre-order

B. Pre-order and Post-order

C. In-order

D. Post-order

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: In-order

All left-subtree keys precede the root, and all right-subtree keys follow it. Recursively visiting left, parent, right therefore emits increasing keys. Evidence: assigned PDF pages 9.

Why each choice is right or wrong

A - Incorrect: Preorder prints root before smaller left keys. See assigned PDF 9, 9.

B - Incorrect: Including preorder cannot guarantee increasing order. See assigned PDF 9, 9.

C - Correct: Inorder matches left keys < root < right keys recursively. See assigned PDF 9, 9.

D - Incorrect: Postorder prints the root after larger right keys. See assigned PDF 9, 9.

Primary note proof

Study-note section DSA19

With distinct keys, every key in a node's left subtree is smaller and every key in its right subtree is larger. Immediate-child comparisons alone are insufficient: root 10, left child 5, and 5's right child 12 violate the root's restriction. Duplicates require a separate count or placement policy.

Study-note section DSA20

Preorder visits node,left,right; inorder left,node,right; postorder left,right,node. Apply recursively; null children do nothing. Inorder returns sorted keys only for a BST. Preorder supports copying, postorder supports deleting children before parents, and level order uses a FIFO queue.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Data Structure and Algorithms module | PDF page 9 | lines 40-44

III. Tree Traversing:
Binary search tree can be traversed in three ways.
a. Pre order traversal - traversing binary tree in the order of parent, left and right.
b. Inorder traversal - traversing binary tree in the order of left, parent and right.
c. Postorder traversal - traversing binary tree in the order of left, right and parent.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 9-4

Computer Science Exit Exam practice collection · Question 137

Question 130 | SM0240

If a sequence of operations: push(1), push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop are performed on a stack, the sequence of popped values is:

A. 2,1,2,2,1

B. 2,2,1,1,2

C. 2,1,2,2,2

D. 2,2,1,2,2

Hint

Trace the initial state and update only the fields affected by each permitted operation.

Answer B: 2,2,1,1,2

Trace bottom-to-top: [1], [1,2], pop 2; [1,1], [1,1,2], pop 2; pop 1; pop 1; [2], pop 2. Popped sequence is 2,2,1,1,2. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: The second pop is 2, not 1. See assigned PDF 6.

B - Correct: Exactly matches all five top removals. See assigned PDF 6.

C - Incorrect: The second pop is again 2, not 1. See assigned PDF 6.

D - Incorrect: The fourth pop removes the original bottom 1, not 2. See assigned PDF 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA12

> Write the stack bottom-to-top after every instruction and record popped values separately. Push 1, push 2, pop emits 2 and leaves [1]. Push 1, push 2, pop, pop, pop then emits 2,1,1; push 2, pop emits 2. The complete output is 2,2,1,1,2.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science Exit Exam · Sene 2016 · Question 13-2

Computer Science Exit Exam practice collection · Question 140

Question 131 | SM0279

Which of the following is FALSE in Data Structures and Algorithms?

A. A tree contains a cycle

B. Tree is a non-linear data structure

C. A tree is a connected graph

D. A tree with n nodes contains (n-1) edges

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: A tree contains a cycle

Unique root paths and one parent per nonroot node exclude cycles. The n-1 nonroot nodes each contribute one parent edge, making the rooted tree connected with n-1 edges. Evidence: assigned PDF pages 9, 2.

Why each choice is right or wrong

A - Correct: A cycle contradicts the source unique-path parent hierarchy. See assigned PDF 9, 2.

B - Incorrect: Trees are explicitly classified as nonlinear. See assigned PDF 9, 2.

C - Incorrect: Every node is reachable from the root along a unique path, so the tree is connected. See assigned PDF 9, 2.

D - Incorrect: One parent edge for each nonroot node yields n-1 edges. See assigned PDF 9, 2.

Primary note proof

Study-note section DSA18

> A rooted tree has one parentless root and one parent per other node. It is connected and acyclic, has n-1 edges, and one path between any pair. Depth counts root-to-node edges; height is maximum depth. A single-node tree therefore has height zero. Leaves have no children.

Study-note section DSA01

> An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 3-13

> A tree is a set of nodes and edges that connect pairs of nodes that connect pairs of nodes. It is an abstract model of a hierarchical structure. Rooted tree has the following structure:
> • One node distinguished as root.
> • Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
> • There is a unique path from the root to the each node.
> • The number of edges in a path is the length of the path.
> Tree Terminologies:
> • Root: a node without a parent.
> • Internal node: a node with at least one child.
> • External (leaf) node: a node without a child.

Data Structure and Algorithms module | PDF page 2 | lines 15-18

a. Linear data structure: data items stores in sequence
Example: Array, stack and Queue,
b. Non-Linear data structures : in which there is order of data items
Example: Trees and Graphs

Question sources

Computer Science Exit Exam · Sene 2016 · Question 26-1

Computer Science Exit Exam practice collection · Question 136

Question 132 | SM0282

A fixed-capacity circular array queue uses front and size indices, modulo slot reuse, and no shifting or resizing. Under unit-cost index arithmetic, what is the tightest listed worst-case pair for enqueue and dequeue, including full/empty checks?

A. O(n), O(1)

B. O(1), O(n)

C. O(1), O(1)

D. O(n), O(n)

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: C. O(1), O(1)

Enqueue computes a slot from front and size, writes one item and updates the count. Dequeue reads the front slot, advances front modulo capacity and decreases the count. Full or empty checks also inspect only constant-size state. No operation shifts or resizes the array, so each uses constant work in the stated model. The linear alternatives are looser bounds rather than the requested tight pair.

Why each choice is right or wrong

A - Incorrect: Linear enqueue is looser than the constant cost of this implementation.

B - Incorrect: Linear dequeue is looser than its constant index update cost.

C - Correct: Both endpoint operations use constant work.

D - Incorrect: Both linear bounds are looser than the requested tight pair.

Primary note proof

Study-note section DSA15

A simple noncircular array queue may report overflow when rear reaches the final index despite released slots at the front. A circular queue reuses slots with modulo arithmetic. Use a count convention: front points to the next removal, size counts items, and insertion slot is (front+size)%C. Empty means size==0; full means size==C.

Study-note section DSA15

Dequeue reads A[front], sets front=(front+1)%C and decreases size. An alternative reserves one unused slot and uses next(rear)==front for full; its empty test and capacity differ. State the convention before tracing.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 18-23

I. Simple Array implementation (static memory).
A simple array structure can be used to implement enqueue & dequeue operations. Thus item will
be added at the end of the array whereas dequeue operation removes items from the front. In simple
array implementation, to add item it checks whether there is room at the rear or not. Therefore, even though there are free released spaces at the front ot the array the queue implementation might
generate overflow message. This is the key drawback to use simple array to implement Queue.

Data Structure and Algorithms module | PDF page 7 | lines 24-26

II. Circular array implementation of Queue Operations
To overcome the above drawback of simple array implementation a circular array can be used, in
which freed spaces are re-used to store data.

Question sources

Computer Science Exit Exam · Sene 2016 · Question 27-1

Computer Science Exit Exam practice collection · Question 141

Question 133 | SS115

Depending on the organization of the elements which of the following approach Is a non-linear data structure?

A. Queues

B. Linked-list

C. Tree

D. Stack

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: Tree

The module classifies trees as nonlinear, while queues and stacks are linear sequences and a linked list is a next-pointer chain. Evidence: assigned PDF pages 2, 4, 6.

Why each choice is right or wrong

A - Incorrect: Queue is an explicit linear example. See assigned PDF 2, 4, 6.

B - Incorrect: A linked list is a chain with one next node, not branching hierarchy. See assigned PDF 2, 4, 6.

C - Correct: Tree is an explicit nonlinear example. See assigned PDF 2, 4, 6.

D - Incorrect: Stack is an explicit linear example. See assigned PDF 2, 4, 6.

Primary note proof

Study-note section DSA01

> An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 15-18

> a. Linear data structure: data items stores in sequence
> Example: Array, stack and Queue,
> b. Non-Linear data structures : in which there is order of data items
> Example: Trees and Graphs

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
> to the chain.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some

useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science model exam collection · Question 22

Question 134 | SS149

For this question, an indirect queue application means using FIFO as auxiliary storage inside a graph traversal algorithm, rather than directly modeling a waiting line or scheduling service. Which listed description fits?

A. Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).

B. Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.

C. Auxiliary data structure for algorithms

D. Multiprogramming

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: C. Auxiliary data structure for algorithms

The supplied convention distinguishes an algorithm's internal work storage from a real-world waiting-line model. BFS uses a queue to hold its discovered but not yet explored vertices, so auxiliary algorithm storage is the matching category. Equal-priority job scheduling and ticket-line simulation directly model service waiting. Multiprogramming is a broader operating-system setting and does not specifically name the internal graph-traversal role requested here.

Why each choice is right or wrong

A - Incorrect: Arrival-order print scheduling directly models a service queue.

B - Incorrect: Ticket-line simulation directly models real-world waiting.

C - Correct: An algorithmic frontier queue is auxiliary storage inside graph traversal.

D - Incorrect: Multiprogramming is a broad execution setting, not the specified internal traversal role.

Primary note proof

Study-note section DSA41

Mark the source when enqueuing it. Repeatedly dequeue a vertex, examine its neighbours, and enqueue each unvisited neighbour while marking it. Store predecessor[v] when discovering v if a path must be reconstructed. FIFO ensures vertices at distance d are explored before later distance layers.

Study-note section DSA14

FIFO suits equal-priority printer jobs, shared-resource waiting, asynchronous buffers and level-order exploration. A play sequence can also use a queue. Distinguish airline seat records from waiting requests; an application name alone does not force one representation.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 41-48

Application of Queue
Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
• Print server- maintains a queue of print jobs
• Disk Driver- maintains a queue of disk input/output requests
• Task scheduler in multiprocessing system- maintains priority queues of processes
• Telephone calls in a busy environment -maintains a queue of telephone calls
• Simulation of waiting line- models real life queues (e.g. supermarkets checkout, customer service line (Bank, custom, airport etc))

Question sources

Computer Science model exam collection · Question 56

Question 135 | SS158

A queue in which the item most recently added is always the first one out refers to:

A. FIFO queue

B. Real time queue

C. Priority queue

D. LIFO queue

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: LIFO queue

Most recently added first removed is LIFO by definition. The name LIFO queue here describes stack discipline rather than the ordinary FIFO queue. Evidence: assigned PDF pages 6, 7, 8.

Why each choice is right or wrong

A - Incorrect: FIFO removes the earliest insertion. See assigned PDF 6, 7, 8.

B - Incorrect: Real time alone does not specify last-in removal. See assigned PDF 6, 7, 8.

C - Incorrect: Priority depends on keys, not necessarily insertion recency. See assigned PDF 6, 7, 8.

D - Correct: LIFO exactly names last-in-first-out service. See assigned PDF 6, 7, 8.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 8 | lines 40-45

> Priority Queue:- is a queue where each data has an associated key provided at insertion time.
> Application of Queue
> Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
> • Print server- maintains a queue of print jobs
> • Disk Driver- maintains a queue of disk input/output requests
> • Task scheduler in multiprocessing system- maintains priority queues of processes

Question sources

Computer Science model exam collection · Question 65

Question 136 | AM0394

Which data structure follows the LIFO (Last-In-First-Out) principle?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: Stack

The stack top is both insertion and removal end, so the most recent unremoved insertion leaves first. Evidence: assigned PDF pages 6, 7, 4, 9.

Why each choice is right or wrong

A - Correct: Stack is explicitly LIFO. See assigned PDF 6, 7, 4, 9.

B - Incorrect: Queue is FIFO. See assigned PDF 6, 7, 4, 9.

C - Incorrect: A linked list is a representation with no mandatory service discipline. See assigned PDF 6, 7, 4, 9.

D - Incorrect: A binary tree defines children, not insertion/removal chronology. See assigned PDF 6, 7, 4, 9.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science MCQ practice collection · Question 394

Computer Science MCQ practice collection · Question 414

Question 137 | AM0395

Which data structure follows the FIFO (First-In-First-Out) principle?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: Queue

FIFO is explicitly the queue policy: enqueue at rear and dequeue at front. Evidence: assigned PDF pages 7, 6.

Why each choice is right or wrong

A - Incorrect: Stack is LIFO. See assigned PDF 7, 6.

B - Correct: Queue preserves insertion order for removal. See assigned PDF 7, 6.

C - Incorrect: A linked list can implement a queue but does not inherently impose FIFO. See assigned PDF 7, 6.

D - Incorrect: Binary-tree structure does not impose FIFO. See assigned PDF 7, 6.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 395

Computer Science MCQ practice collection · Question 415

Question 138 | AM0396

Which listed representation permits insertion and removal at either endpoint in constant time? Use ordinary stack and FIFO queue interfaces, and interpret C as a doubly linked list maintaining both head and tail.

A. Stack

B. Queue

C. Doubly linked list with head and tail

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: C. Doubly linked list with head and tail

A doubly linked list stores predecessor and successor links, while head and tail provide direct access to both endpoints. Inserting or removing there updates a fixed number of links, including the empty and singleton cases. Ordinary stack and FIFO queue interfaces expose fewer endpoint operations, and a binary tree has no corresponding two-end sequence guarantee. The explicit linked-list variant is essential because singly linked tail deletion needs a predecessor traversal.

Why each choice is right or wrong

A - Incorrect: A stack exposes insertion/removal at one top only.

B - Incorrect: An ordinary FIFO queue does not permit all four endpoint operations.

C - Correct: A doubly linked list with both endpoint pointers supports constant-time operations at each end.

D - Incorrect: A binary tree has no two-end sequence-operation guarantee.

Primary note proof

Study-note section DSA10

> Doubly linked nodes add prev; head and tail enable constant-time operations at both ends. Singly linked tail deletion still needs its predecessor. A circular tail links back to head; stop traversal on returning to the start, handling empty separately.

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
> to the chain.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

> Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
> Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Question sources

Computer Science MCQ practice collection · Question 396

Question 139 | AM0397

Which data structure represents a hierarchical structure with nodes and edges?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer D: Binary tree

A tree is expressly an abstract model of a hierarchical structure, and a binary tree is its at-most-two-children specialization. Evidence: assigned PDF pages 9.

Why each choice is right or wrong

A - Incorrect: Stack is a one-ended linear list. See assigned PDF 9, 9.

B - Incorrect: Queue is a FIFO linear list. See assigned PDF 9, 9.

C - Incorrect: A singly linked list is a chain rather than a branching hierarchy. See assigned PDF 9, 9.

D - Correct: Binary tree matches the hierarchy with nodes and edges. See assigned PDF 9, 9.

Primary note proof

Study-note section DSA18

> A rooted tree has one parentless root and one parent per other node. It is connected and acyclic, has n-1 edges, and one path between any pair. Depth counts root-to-node edges; height is maximum depth. A single-node tree therefore has height zero. Leaves have no children.

Study-note section DSA01

> Trees model hierarchies; graphs model general connections.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 3-13

> A tree is a set of nodes and edges that connect pairs of nodes that connect pairs of nodes. It is an abstract model of a hierarchical structure. Rooted tree has the following structure:
> • One node distinguished as root.
> • Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
> • There is a unique path from the root to the each node.
> • The number of edges in a path is the length of the path.
> Tree Terminologies:

• Root: a node without a parent.
• Internal node: a node with at least one child.
• External (leaf) node: a node without a child.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science MCQ practice collection · Question 397

Question 140 | AM0398

Which data structure is best suited for implementing a backtracking algorithm?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: Stack

Backtracking is explicitly listed as a stack application: the latest pending decision can be undone first using pop. Evidence: assigned PDF pages 7, 6.

Why each choice is right or wrong

A - Correct: LIFO naturally returns to the most recent unfinished decision. See assigned PDF 7, 6.

B - Incorrect: FIFO would return to oldest pending work. See assigned PDF 7, 6.

C - Incorrect: A generic linked list does not prescribe backtracking order. See assigned PDF 7, 6.

D - Incorrect: A binary tree can represent choices but does not itself maintain a LIFO return path. See assigned PDF 7, 6.

Primary note proof

Study-note section DSA12

Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 398

Question 141 | AM0399

A dispatcher keeps pending jobs in a sequence. Urgent jobs are added at the front, ordinary jobs at the back; serving removes the front job, and cancellation removes the last job. Which listed representation supports every operation in worst-case constant time as the sequence grows? Use ordinary stack and FIFO queue interfaces, with no additional indexes or auxiliary structures.

A. Stack

B. Queue

C. Doubly linked list with head and tail

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: C. Doubly linked list with head and tail

Urgent arrival inserts at the head, ordinary arrival appends at the tail, service removes the head, and cancellation removes the tail. A doubly linked list with both endpoint pointers updates a fixed number of links for each action; its prev link identifies the predecessor needed for tail removal. A stack exposes only one top, a FIFO queue lacks last-job cancellation and front insertion, and a generic binary tree provides no such endpoint-cost guarantee.

Why each choice is right or wrong

A - Incorrect: One stack top cannot directly supply both service-at-front and cancellation-at-back operations.

B - Incorrect: An ordinary FIFO queue permits rear insertion and front removal, but not urgent front insertion or last-job removal.

C - Correct: Head and tail permit direct endpoint access, and prev links make last-job removal constant time.

D - Incorrect: A generic binary tree has no guarantee for all four ordered-sequence endpoint updates.

Primary note proof

Study-note section DSA10

> Doubly linked nodes add prev; head and tail enable constant-time operations at both ends. Singly linked tail deletion still needs its predecessor. A circular tail links back to head; stop traversal on returning to the start, handling empty separately.

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node

added
to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 13-24

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.
if (start == NULL) // If list is empty, then new node become start node
start = temp;
else {
temp2 = start; // list not empty! Thus temp2 starts from first node
while (temp2->next!= NULL) { // till the current last node found navigates forward
temp2 = temp2->next; // Move to next link in chain
}
temp2->next = temp; // new node (temp) appended at the end of the list

Data Structure and Algorithms module | PDF page 6 | lines 4-13

2. Make temp1 point to whatever the start pointer is pointing to.
3. If the next pointer of what temp1 indicates is NULL, then we've found the last node of the list, so
jump to step 7.
4. Make another pointer, temp2, point to the current node in the list.
5. Make temp1 point to the next item in the list.
6. Go to step 3.
7. If you get this far, then the temporary pointer, temp1, should point to the last item in the list and
the other temporary pointer, temp2, should point to the last-but-one item.
8. Delete the node pointed to by temp1.
9. Mark the next pointer of the node pointed to by temp2 as NULL - it is the new last node.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Question sources

Computer Science MCQ practice collection · Question 399

Question 142 | AM0400

Which listed structures admit both array-based and linked-node representations? Links may be array indices or pointers, and binary-tree representations include the complete binary-tree case. Select all that apply.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A, B, C, D: A. Stack; B. Queue; C. Linked list; D. Binary tree

Stacks and queues have both array and linked-node implementations. A linked list can store its next links as indices into an array or as pointers between allocated nodes. A binary tree can use linked child references, while the complete binary-tree case has the indexed heap representation. The question concerns possible representations, so all four qualify without requiring one universal layout for every instance.

Why each choice is right or wrong

A - Correct: A stack can use an array top or a linked head.

B - Correct: A queue can use circular slots or linked endpoint nodes.

C - Correct: List links may be indices or node references.

D - Correct: Binary trees have linked-node forms and indexed complete-tree forms.

Primary note proof

Study-note section DSA01

> Arrays or linked nodes can implement the same LIFO stack.

Study-note section DSA14

> A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head.

Study-note section DSA15

> A circular queue reuses slots with modulo arithmetic.

Study-note section DSA08

> An array-linked list still follows links unless separately position-indexed.

Study-note section DSA23

> Zero-based array representation: root index 0, left=2i+1, right=2i+2, parent=floor((i-1)/2) for i>0.

Study-note section DSA-S03

A linked binary-tree node stores a payload and two child pointers, one for the left child and one for the right child.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

I. Array implementation (static memory).
Push(n): Adding item at top
top is the current top of stack and n is its
maximum size
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;

Data Structure and Algorithms module | PDF page 6 | lines 43-45

II. Linked list (dynamic memory)
The linked list implementation of stack is adding items at the end of the chain (Push item) and deleting the end node of the list (Pop item).

Data Structure and Algorithms module | PDF page 7 | lines 18-23

I. Simple Array implementation (static memory).
A simple array structure can be used to implement enqueue & dequeue operations. Thus item will
be added at the end of the array whereas dequeue operation removes items from the front. In simple
array implementation, to add item it checks whether there is room at the rear or not. Therefore, even though there are free released spaces at the front ot the array the queue implementation might
generate overflow message. This is the key drawback to use simple array to implement Queue.

Data Structure and Algorithms module | PDF page 8 | lines 35-37

III. Linked list implementation of enqueue and dequeue operations
Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

Question sources

Computer Science MCQ practice collection · Question 400

Computer Science MCQ practice collection · Question 418

Question 143 | AM0401

A character reader supplies a word from left to right. After reading finishes, an output routine must remove one character at a time from a single auxiliary ADT to display the word right to left. It may use only the ADT's standard insertion/removal interface and must preserve the original input. Which ADT directly supplies the needed removal order?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: A. Stack

Insert the characters in the order they are read. The required output starts with the last character inserted, then continues toward the earliest one, so a stack's LIFO interface directly supplies the reversal. A queue would preserve the original order. A linked list could represent a stack, but its storage links alone do not prescribe LIFO; a generic tree likewise has no matching removal contract.

Why each choice is right or wrong

A - Correct: Repeated stack removals produce the characters in reverse insertion order.

B - Incorrect: FIFO removal would reproduce the original character order.

C - Incorrect: A list is a possible representation but does not itself prescribe the required removal policy.

D - Incorrect: A generic tree does not define the requested reverse-insertion removal contract.

Primary note proof

Study-note section DSA12

Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Study-note section DSA11

Stack policy is LIFO, equivalently FILO.

Study-note section DSA11

Linked stacks achieve constant time by inserting and deleting at the head; using a singly linked tail for pop would require a predecessor traversal.

Study-note section DSA14

An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served.

Study-note section DSA45

Specify whether a question asks which structure can implement something or which is best under stated costs. ADTs do not force one representation, and serial-access memory is not inherently LIFO.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 401

Computer Science MCQ practice collection · Question 424

Question 144 | AM0402

Which data structure is commonly used for implementing function call stacks in programming languages?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A: Stack

The source explicitly states that a memory stack is used in function calls including recursion, keeping the most recent active call on top. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Correct: Nested calls return in LIFO order through a stack. See assigned PDF 6, 6.

B - Incorrect: Queue FIFO order would not reflect nested returns. See assigned PDF 6, 6.

C - Incorrect: Linked nodes can store frames but the controlling ADT is still a stack. See assigned PDF 6, 6.

D - Incorrect: A binary tree does not provide the specified call-return discipline. See assigned PDF 6, 6.

Primary note proof

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 46-52

> Applications of Stack
> • It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
> • Infix to Postfix Transformation
> • It is useful during the execution of recursive programs
> • A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
> • A stack (memory stack) can be used in function calls including recursion.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

"A stack is an ordered list in which all insertions and deletions are made at one end, called the top". Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 402

Question 145 | AM0403

A message buffer accepts new items at its rear and delivers the oldest waiting item from its front. Which service ADT can maintain this contract using either a circular array or linked head-and-tail nodes? Choose the ADT rather than an underlying storage container.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: B. Queue

The oldest waiting message must leave first, while new arrivals join the rear. This is the ordinary queue service contract. The note supplies both a circular-array representation that reuses slots and a linked representation that appends at the tail and removes at the head. A stack serves the newest item first; a generic linked list or tree does not independently specify this FIFO policy.

Why each choice is right or wrong

A - Incorrect: A stack's latest-first removal violates oldest-message-first delivery.

B - Correct: A queue has the stated rear-insertion/front-removal contract and both cited implementations.

C - Incorrect: Linked nodes can store the queue but do not by themselves define its service policy.

D - Incorrect: A binary tree does not inherently define this FIFO message-buffer interface.

Primary note proof

Study-note section DSA15

A simple noncircular array queue may report overflow when rear reaches the final index despite released slots at the front. A circular queue reuses slots with modulo arithmetic. Use a

count convention: front points to the next removal, size counts items, and insertion slot is (front+size)%C. Empty means size==0; full means size==C.

Study-note section DSA14

An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 24-26

II. Circular array implementation of Queue Operations
To overcome the above drawback of simple array implementation a circular array can be used, in
which freed spaces are re-used to store data.

Data Structure and Algorithms module | PDF page 8 | lines 35-37

III. Linked list implementation of enqueue and dequeue operations
Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

Question sources

Computer Science MCQ practice collection · Question 403

Question 146 | AM0404

Which data structure has a First-In-First-Out (FIFO) property?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: Queue

First-in-first-out is the explicit queue property. Inserting at rear and removing at front preserves the oldest pending element. Evidence: assigned PDF pages 7.

Why each choice is right or wrong

A - Incorrect: Stack is last-in-first-out. See assigned PDF 7.

B - Correct: Queue has FIFO semantics. See assigned PDF 7.

C - Incorrect: A linked-list representation alone imposes no FIFO restriction. See assigned PDF 7.

D - Incorrect: A binary-tree hierarchy is not arrival-order access. See assigned PDF 7.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Computer Science MCQ practice collection · Question 404

Question 147 | AM0405

Which data structure is suitable for implementing a breadth-first search algorithm?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer B: Queue

DSA41 enqueues discovered neighbors and dequeues the next pending vertex. This FIFO discipline preserves distance-layer ordering. A linked list can store the queue but does not by itself state the policy. Stack service is latest-first, while a binary tree is not the standard frontier discipline. Queue is therefore the specific suitable ADT among the choices.

Why each choice is right or wrong

A - Incorrect: Stack order selects the latest pending vertex and does not preserve the BFS layer invariant.

B - Correct: Queue FIFO ordering processes earlier distance layers before later discoveries, as BFS requires.

C - Incorrect: A linked list may represent a queue, but linked-list links alone specify no BFS service order.

D - Incorrect: A binary tree does not supply the standard FIFO pending-vertex discipline used by breadth-first search.

Primary note proof

Study-note section DSA41

> Mark the source when enqueuing it. Repeatedly dequeue a vertex, examine its neighbours, and enqueue each unvisited neighbour while marking it. Store predecessor[v] when discovering v if a path must be reconstructed. FIFO ensures vertices at distance d are explored before later distance layers.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.

Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Computer Science MCQ practice collection · Question 405

Computer Science MCQ practice collection · Question 426

Question 148 | AM0406

Which listed structures can create nodes using dynamic memory allocation in a linked-node implementation? Select all that apply.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A, B, C, D: A. Stack; B. Queue; C. Linked list; D. Binary tree

Dynamic node allocation is a representation choice rather than a property exclusive to one ADT. A linked stack can allocate a node for each push, a linked queue for each enqueue, and a linked list for insertion. A pointer-based binary tree can allocate a leaf when attaching it. Array implementations also exist, but their existence does not prevent any of these structures from using dynamically allocated nodes.

Why each choice is right or wrong

A - Correct: A linked stack can allocate each new head node.

B - Correct: A linked queue can allocate each new rear node.

C - Correct: A linked list can grow by allocating nodes as needed.

D - Correct: A linked binary tree can allocate an inserted leaf.

Primary note proof

Study-note section DSA-S03

A linked binary-tree node stores a payload and two child pointers, one for the left child and one for the right child. A null child pointer means that child is absent; a root pointer gives access to the whole tree. In a dynamically allocated version, insertion can allocate a node and attach it through the appropriate child link.

Study-note section DSA08

Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Study-note section DSA11

Linked stacks achieve constant time by inserting and deleting at the head;

Study-note section DSA14

A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 5 | lines 3-8

While adding a node to a list, the chain needs to be rearranged properly so that the each nodes are connected to each other. It is possible to add a node anywhere in the chain (at the start, middle or end). Let's consider adding a node at the end of the list:
Step 1: First we need to reserve a space for the new node using the keyword new.
Node *temp = new *Node; // Reserving a space for temporary pointer which points to new node;

Data Structure and Algorithms module | PDF page 6 | lines 43-45

II. Linked list (dynamic memory)
The linked list implementation of stack is adding items at the end of the chain (Push item) and deleting the end node of the list (Pop item).

Data Structure and Algorithms module | PDF page 8 | lines 35-37

III. Linked list implementation of enqueue and dequeue operations
Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science MCQ practice collection · Question 406

Question 149 | AM0407

Which structures admit an array-based representation? Array slots may contain payloads and integer links; for binary trees include complete trees stored with child-index formulas. Select all that apply.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer A, B, C, D: A. Stack; B. Queue; C. Linked list; D. Binary tree

An array stack uses a top index, while a circular queue uses front and size indices. A linked list can store its links as indices into an array rather than as pointers to separately allocated nodes. Complete binary trees also fit the indexed heap layout. Since the question asks whether representations exist and removes the subjective ease ranking, each of the four listed structures qualifies.

Why each choice is right or wrong

A - Correct: An array top index implements a stack.

B - Correct: Modulo-indexed slots implement a circular queue.

C - Correct: Array indices can encode list links.

D - Correct: Complete binary trees use the indexed heap layout.

Primary note proof

Study-note section DSA11

> With zero-based occupied slots, top=-1 is empty and top=C-1 is full.

Study-note section DSA15

> insertion slot is (front+size)%C.

Study-note section DSA08

> An array-linked list still follows links unless separately position-indexed.

Study-note section DSA23

> Zero-based array representation: root index 0, left=2i+1, right=2i+2, parent=floor((i-1)/2) for i>0. One-based representation uses children 2i,2i+1 and parent floor(i/2). A child exists only if its index is inside the current heap size.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

I. Array implementation (static memory).
Push(n): Adding item at top
top is the current top of stack and n is its
maximum size
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;

Data Structure and Algorithms module | PDF page 7 | lines 18-23

I. Simple Array implementation (static memory).
A simple array structure can be used to implement enqueue & dequeue operations. Thus item will
be added at the end of the array whereas dequeue operation removes items from the front. In simple
array implementation, to add item it checks whether there is room at the rear or not. Therefore, even though there are free released spaces at the front ot the array the queue implementation might
generate overflow message. This is the key drawback to use simple array to implement Queue.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.

Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Computer Science MCQ practice collection · Question 407

Question 150 | AM0408

Which listed representation provides direct next and previous links for walking a linear sequence forward and backward without retracing from its head? Use ordinary stack/queue interfaces and a binary tree without parent or traversal-thread links.

A. Stack

B. Queue

C. Doubly linked list

D. Binary tree

Hint

Compare the required operations, access order and representation metadata with each option.

Answer C: C. Doubly linked list

A doubly linked list retains the forward next link and adds a prev link to each node. Following the corresponding link moves directly to a neighboring node in either direction. Ordinary stack and queue interfaces do not expose this bidirectional traversal contract. A child-linked binary tree without parent or traversal threads also does not provide the specified previous/next linear-sequence links. The concrete list variant makes the guarantee precise.

Why each choice is right or wrong

A - Incorrect: A stack interface exposes its top, not a bidirectional traversal.

B - Incorrect: A queue interface exposes FIFO endpoints, not neighboring prev/next traversal.

C - Correct: Doubly linked nodes provide next and prev links.

D - Incorrect: The specified child-only tree lacks the requested linear previous/next links.

Primary note proof

Study-note section DSA10

Doubly linked nodes add prev; head and tail enable constant-time operations at both ends. Singly linked tail deletion still needs its predecessor. A circular tail links back to head; stop traversal on returning to the start, handling empty separately.

Study-note section DSA08

A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Computer Science MCQ practice collection · Question 408

Question 151 | AM0409

Which specified representations provide direct constant-time inspection of both first and last items in a nonempty sequence? A is a head-only linked stack, B a linked queue with front and rear pointers, C a linked list with head and tail, and D a child-linked binary tree with no cached endpoints. Select all that apply.

A. Head-only linked stack

B. Linked queue with front and rear

C. Linked list with head and tail

D. Child-linked binary tree without endpoint caches

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B, C: B. Linked queue with front and rear; C. Linked list with head and tail

Maintaining references to both endpoints makes their items directly available without traversal. The specified linked queue retains front and rear, and the specified list retains head and tail, so both provide constant-time inspection. A head-only linked stack must follow links to reach its oldest item, and a generic child-linked tree without endpoint caches supplies no direct first/last guarantee. The answer follows the representations, not the generic names alone.

Why each choice is right or wrong

A - Incorrect: A head-only stack lacks a direct reference to its oldest node.

B - Correct: Front and rear pointers directly expose both endpoint items.

C - Correct: Head and tail pointers directly expose both endpoint items.

D - Incorrect: The unaugmented tree has no cached traversal endpoints.

Primary note proof

Study-note section DSA14

> A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head.

Study-note section DSA10

> Doubly linked nodes add prev; head and tail enable constant-time operations at both ends.

Study-note section DSA11

> Linked stacks achieve constant time by inserting and deleting at the head;

Study-note section DSA-S03

> A linked binary-tree node stores a payload and two child pointers, one for the left child and one for the right child.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow

to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 13-24

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.

```
if (start == NULL) // If list is empty, then new node become start node
start = temp;
else {
temp2 = start; // list not empty! Thus temp2 starts from first node
while (temp2->next!= NULL) { // till the current last node found navigates forward
temp2 = temp2->next; // Move to next link in chain
}
temp2->next = temp; // new node (temp) appended at the end of the list
```

Question sources

Computer Science MCQ practice collection · Question 409

Question 152 | AM0410

Which data structure can be efficiently used for implementing undo and redo functionality in applications?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: Stack

Undo reverses the most recent action first, matching the source stack backtracking/reversal uses. A second stack can retain undone actions and reverse them again for redo. Evidence: assigned PDF pages 7, 6.

Why each choice is right or wrong

A - Correct: Push records actions; pop undoes the most recent; a second LIFO stack supports redo. See assigned PDF 7, 6.

B - Incorrect: FIFO would undo the oldest pending action first. See assigned PDF 7, 6.

C - Incorrect: A generic linked list can implement a stack but supplies no undo discipline by itself. See assigned PDF 7, 6.

D - Incorrect: A binary tree does not directly supply last-action-first reversal. See assigned PDF 7, 6.

Primary note proof

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

> • Reversing Data and Reverse a List
> • Convert Decimal to Binary
> • Parsing - It is a logic that breaks into independent pieces for further processing
> • Backtracking
> Examples:
> 1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
> 2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 410

Question 153 | AM0411

A forward-only input sequence must be emitted in reverse order after all input is read, using one auxiliary ADT through its standard insertion/removal interface. Which ADT directly reverses the order without modifying the input links or adding a separate reversal algorithm?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: A. Stack

A stack accepts the sequence in arrival order and then removes the most recently inserted item first. Repeated pops therefore emit the complete reverse sequence directly. A FIFO queue preserves the original order. A generic linked list or binary tree does not itself specify the required removal discipline; reversing existing links would be a separate algorithm excluded by this question. The contract identifies the auxiliary reversal ADT.

Why each choice is right or wrong

A - Correct: Push all items and then pop to emit reverse order directly.

B - Incorrect: FIFO removal preserves input order.

C - Incorrect: A generic list has no inherent reverse-removal contract without an added algorithm or stack policy.

D - Incorrect: A generic tree supplies no required reverse-sequence service.

Primary note proof

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 3-8

While adding a node to a list, the chain needs to be rearranged properly so that the each nodes are connected to each other. It is possible to add a node anywhere in the chain (at the start, middle or end). Let’s consider adding a node at the end of the list:
Step 1: First we need to reserve a space for the new node using the keyword new.
Node *temp = new *Node; // Reserving a space for temporary pointer which points to new node;

Question sources

Computer Science MCQ practice collection · Question 411

Question 154 | AM0412

Which abstract frontier service policy directly implements the latest-pending-work-first behavior of depth-first search? Choose the ADT, rather than a storage representation that could be given that policy.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: A. Stack

DFS finishes the most recently entered unfinished branch before returning to older pending work. A stack directly supplies this latest-pending-work-first rule. A queue would instead remove older pending items first, producing a breadth-first discipline under ordinary graph traversal. A linked list may implement a stack, but its links alone do not prescribe the frontier policy. A binary tree describes possible input relationships rather than the required pending-work ADT.

Why each choice is right or wrong

A - Correct: A stack directly supplies latest-pending-work-first service.

B - Incorrect: A FIFO queue retrieves older pending work first.

C - Incorrect: Linked-list storage alone does not specify a frontier service policy.

D - Incorrect: A binary tree is not the latest-pending-work service ADT.

Primary note proof

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

> • Reversing Data and Reverse a List
> • Convert Decimal to Binary
> • Parsing - It is a logic that breaks into independent pieces for further processing
> • Backtracking

Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Question sources

Computer Science MCQ practice collection · Question 412

Question 155 | AM0413

Which listed representations can implement a priority queue without retaining an unmodified LIFO or FIFO removal contract? Interpret D as a binary heap and C as an ordered or unordered list using extremum selection. Select all that apply.

A. Stack

B. Queue

C. Linked list

D. Binary tree organized as a binary heap

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C, D: C. Linked list; D. Binary tree organized as a binary heap

A priority queue removes an item with an extreme priority key. An unordered list can scan for that extremum, while an ordered list maintains the relevant order during insertion. A binary heap supplies the same priority contract through heap order and logarithmic insertion/extraction. An unmodified stack or FIFO queue removes according to arrival position rather than priority. Thus the list and heap-shaped binary tree qualify under the stated algorithms.

Why each choice is right or wrong

A - Incorrect: An unmodified stack removes by recency rather than priority.

B - Incorrect: An unmodified FIFO queue removes by arrival order rather than priority.

C - Correct: Ordered or scanned lists can select the extreme priority.

D - Correct: A binary heap implements priority selection through heap order.

Primary note proof

Study-note section DSA17

A priority queue returns the smallest or largest priority key under its convention. Urgent patients may precede earlier arrivals. Insertions and extractions can interleave. Binary heaps support constant peek and logarithmic insert/extract.

Study-note section DSA17

Arrays and lists can also implement priority queues: an unordered sequence gives cheap insertion but linear extremum removal; an ordered sequence reverses that tradeoff. Equal priorities have no inherent FIFO guarantee. If stable service is required, compare (priority, arrival_counter). Choose the priority direction and tie rule before tracing.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 40-45

Priority Queue:- is a queue where each data has an associated key provided at insertion time.
Application of Queue
Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
• Print server- maintains a queue of print jobs
• Disk Driver- maintains a queue of disk input/output requests
• Task scheduler in multiprocessing system- maintains priority queues of processes

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Question sources

Computer Science MCQ practice collection · Question 413

Question 156 | AM0416

Which ordinary ADT restricts insertion to the rear and removal or peek to the front, preserving arrival order? Assume standard ADT interfaces rather than additional traversal operations.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B: B. Queue

The ordinary queue inserts new items at its rear and removes or observes the front item. These restrictions produce FIFO service: earlier arrivals are removed before later arrivals. A stack uses a single top and LIFO order. A general linked list permits operations beyond these restricted endpoints, while a binary tree organizes hierarchical relationships. Specifying the exact operations resolves the original vague use of access.

Why each choice is right or wrong

A - Incorrect: A stack operates at one top using LIFO.

B - Correct: A queue has rear insertion and front removal/peek.

C - Incorrect: A general list does not restrict access to those two operations.

D - Incorrect: A binary tree does not have this FIFO endpoint contract.

Primary note proof

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

> Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
> Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Question sources

Computer Science MCQ practice collection · Question 416

Question 157 | AM0417

Which structures can use nodes connected by pointers in a linked implementation? Do not require that every implementation use pointers. Select all that apply.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A, B, C, D: A. Stack; B. Queue; C. Linked list; D. Binary tree

Pointer links can connect the nodes of several structures. A linked list uses next links, a linked stack operates at the chain head, and a linked queue maintains endpoint references around a chain. A pointer-based binary tree instead links child nodes along branches. All four therefore admit a node-and-pointer representation. Their alternative array layouts do not invalidate this existence claim about linked implementations.

Why each choice is right or wrong

A - Correct: A linked stack connects nodes through head-chain links.

B - Correct: A linked queue connects nodes and retains endpoint pointers.

C - Correct: A linked list directly uses node links.

D - Correct: A pointer-based tree links its child nodes.

Primary note proof

Study-note section DSA-S03

A linked binary-tree node stores a payload and two child pointers, one for the left child and one for the right child. A null child pointer means that child is absent; a root pointer gives access to the whole tree. In a dynamically allocated version, insertion can allocate a node and attach it through the appropriate child link.

Study-note section DSA01

Arrays or linked nodes can implement the same LIFO stack.

Study-note section DSA14

A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head.

Study-note section DSA08

A singly linked node contains data and next.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science MCQ practice collection · Question 417

Question 158 | AM0419

Which data structure allows elements to be inserted and removed from both ends?

A. Stack

B. Queue

C. Deque

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Deque

A double-ended queue permits insertion and deletion at either end by definition. Evidence: assigned PDF pages 8.

Why each choice is right or wrong

A - Incorrect: Stack restricts both operations to one top. See assigned PDF 8.

B - Incorrect: Ordinary queue fixes insertion rear and deletion front. See assigned PDF 8.

C - Correct: Deque is expressly defined by operations at both ends. See assigned PDF 8.

D - Incorrect: A binary tree is a hierarchy, not a double-ended sequence. See assigned PDF 8.

Primary note proof

Study-note section DSA16

> A deque permits both-end insertion/removal without guaranteeing middle access. Endpoint-linked operations are constant but middle indexing is linear. A circular-array deque may support indexed access. Dequeue names removal; deque names the structure.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

> Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
> Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Question sources

Computer Science MCQ practice collection · Question 419

Question 159 | AM0420

Which ADTs impose an item-removal order through their ordinary service contract? Use LIFO for stacks, FIFO for queues, and smallest-priority-first with FIFO ties for priority queues. A generic binary tree has no specified traversal. Select all that apply.

A. Stack

B. Queue

C. Priority queue

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A, B, C: A. Stack; B. Queue; C. Priority queue

Stacks impose reverse arrival order and queues impose arrival order. A priority queue also imposes an order once the priority direction and tie rule are supplied; this version chooses the smallest priority and serves equal priorities FIFO. A generic binary tree supplies a shape with several possible traversals rather than a single inherent removal order. Consequently the three service ADTs qualify and the unspecified tree does not.

Why each choice is right or wrong

A - Correct: A stack imposes LIFO removal.

B - Correct: A queue imposes FIFO removal.

C - Correct: The stated priority and tie rules define priority-queue removal order.

D - Incorrect: A generic tree without a traversal has no single inherent removal order.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA14

> This is FIFO or first-come, first-served.

Study-note section DSA17

> Arrays and lists can also implement priority queues: an unordered sequence gives cheap insertion but linear extremum removal; an ordered sequence reverses that tradeoff. Equal priorities have no inherent FIFO guarantee. If stable service is required, compare (priority, arrival_counter). Choose the priority direction and tie rule before tracing.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 8 | lines 40-45

> Priority Queue:- is a queue where each data has an associated key provided at insertion time.
> Application of Queue
> Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
> • Print server- maintains a queue of print jobs
> • Disk Driver- maintains a queue of disk input/output requests
> • Task scheduler in multiprocessing system- maintains priority queues of processes

Data Structure and Algorithms module | PDF page 9 | lines 40-44

> III. Tree Traversing:
> Binary search tree can be traversed in three ways.
> a. Pre order traversal - traversing binary tree in the order of parent, left and right.
> b. Inorder traversal - traversing binary tree in the order of left, parent and right.
> c. Postorder traversal - traversing binary tree in the order of left, right and parent.

Question sources

Computer Science MCQ practice collection · Question 420

Question 160 | AM0421

Which is the most general listed binary-tree family defined by the following distinct-key invariant alone, without an additional balancing discipline: every key in a node's left subtree is smaller, and every key in its right subtree is larger?

A. AVL tree

B. Binary search tree

C. Red-black tree

D. B-tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B: B. Binary search tree

The global left-smaller/right-larger invariant defines the binary search tree family. It applies to every descendant in each subtree, not only immediate children. AVL and red-black trees satisfy that ordering but impose additional balancing disciplines, so they are more specialized than the family requested. The binary-tree restriction also excludes a general B-tree. Asking for the most general family prevents those BST subclasses from becoming competing selections.

Why each choice is right or wrong

A - Incorrect: AVL adds balance requirements beyond the stated invariant.

B - Correct: BST is the general family defined by the strict subtree ordering alone.

C - Incorrect: Red-black trees add a balancing discipline beyond the stated invariant.

D - Incorrect: A general B-tree is not the requested binary-tree family.

Primary note proof

Study-note section DSA19

> With distinct keys, every key in a node's left subtree is smaller and every key in its right subtree is larger. Immediate-child comparisons alone are insufficient: root 10, left child 5, and 5's right child 12 violate the root's restriction. Duplicates require a separate count or placement policy.

Study-note section DSA22

> AVL balance factor is height(left)-height(right); every node must have factor -1,0 or 1. After insertion or deletion, update heights upward and rotate where necessary. LL imbalance needs a right rotation; RR needs left; LR rotates the left child left then the node right; RL is symmetric. Rotations preserve inorder key order.

Study-note section DSA-S03

> A B-tree is a balanced multiway search tree. Its allowed branching factor can exceed two, so a general B-tree is not the binary-tree family. Balanced describes a height discipline; it does not by itself restrict every node to two children.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science MCQ practice collection · Question 421

Question 161 | AM0422

Which data structure represents a hierarchical structure with a set of connected nodes?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer D: Binary tree

The module defines trees as hierarchical structures consisting of nodes and connecting edges. Binary trees specialize that hierarchy to two possible children. Evidence: assigned PDF pages 9.

Why each choice is right or wrong

A - Incorrect: Stack is a linear LIFO list. See assigned PDF 9, 9.

B - Incorrect: Queue is a linear FIFO list. See assigned PDF 9, 9.

C - Incorrect: Single linked list is a sequential chain. See assigned PDF 9, 9.

D - Correct: Binary tree is the hierarchical option. See assigned PDF 9, 9.

Primary note proof

Study-note section DSA18

A rooted tree has one parentless root and one parent per other node. It is connected and acyclic, has n-1 edges, and one path between any pair. Depth counts root-to-node edges; height is maximum depth. A single-node tree therefore has height zero. Leaves have no children.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 3-13

A tree is a set of nodes and edges that connect pairs of nodes that connect pairs of nodes. It is an abstract model of a hierarchical structure. Rooted tree has the following structure:
• One node distinguished as root.
• Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
• There is a unique path from the root to the each node.
• The number of edges in a path is the length of the path.
Tree Terminologies:
• Root: a node without a parent.
• Internal node: a node with at least one child.
• External (leaf) node: a node without a child.

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science MCQ practice collection · Question 422

Question 162 | AM0423

Which data structure allows efficient insertion and deletion at both ends?

A. Stack

B. Queue

C. Deque

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Deque

The source deque definition allows insertion and deletion at either end. With endpoint references these are local operations; no middle search is needed. Evidence: assigned PDF pages 8.

Why each choice is right or wrong

A - Incorrect: Stack offers only the top endpoint. See assigned PDF 8.

B - Incorrect: Queue fixes rear insertion and front deletion, not both operations at both ends. See assigned PDF 8.

C - Correct: Deque supplies the requested two-ended operations. See assigned PDF 8.

D - Incorrect: Binary trees do not define two list ends. See assigned PDF 8.

Primary note proof

Study-note section DSA16

> A deque permits both-end insertion/removal without guaranteeing middle access. Endpoint-linked operations are constant but middle indexing is linear. A circular-array deque may support indexed access. Dequeue names removal; deque names the structure.

Study-note section DSA10

> Doubly linked nodes add prev; head and tail enable constant-time operations at both ends. Singly linked tail deletion still needs its predecessor. A circular tail links back to head; stop traversal on returning to the start, handling empty separately.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

> Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
> Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Question sources

Computer Science MCQ practice collection · Question 423

Question 163 | AM0425

Which data structure uses the concept of "first in, last out" to handle function calls and recursion?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: Stack

FILO is explicitly equivalent to LIFO, and the source names stack memory for calls and recursion. The most recent call returns before the earlier waiting caller. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Correct: Stack supplies FILO nesting for function frames. See assigned PDF 6, 6.

B - Incorrect: Queue is FIFO and would not match nested returns. See assigned PDF 6, 6.

C - Incorrect: A linked list can store frames but does not itself impose call order. See assigned PDF 6, 6.

D - Incorrect: A binary tree does not implement the stated endpoint discipline by its definition. See assigned PDF 6, 6.

Primary note proof

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 6 | lines 46-52

> Applications of Stack
> • It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
> • Infix to Postfix Transformation
> • It is useful during the execution of recursive programs

• A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
• A stack (memory stack) can be used in function calls including recursion.

Question sources

Computer Science MCQ practice collection · Question 425

Question 164 | AM0427

Which data structure is suitable for implementing depth-first search (DFS) algorithm?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: Stack

DSA42 describes recursively finishing an unvisited neighbor before returning to the parent. Explicit stack storage or the runtime call stack preserves this latest-pending-work-first behavior. A FIFO queue changes the exploration order. Linked lists can store frames but do not themselves define stack order, and a binary tree alone is not a frontier policy.

Why each choice is right or wrong

A - Correct: A stack stores pending frames or vertices so the latest unfinished branch is resumed first.

B - Incorrect: A queue serves the oldest pending work and changes the depth-first exploration discipline.

C - Incorrect: A linked list may store a stack, but its name alone supplies no depth-first service order.

D - Incorrect: A binary tree is not the standard pending-work ADT needed to preserve DFS return points.

Primary note proof

Study-note section DSA42

Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 427

Question 165 | AM0428

A capacity-limited cache evicts entries strictly in insertion order; cache hits do not change that order. Which ADT should maintain its eviction order? Ask for the service policy rather than a possible storage implementation.

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B: B. Queue

The earliest inserted resident entry must be evicted first, and hits do not move it. This is FIFO order, which a queue directly maintains through rear insertion and front removal. A linked list could be used to implement that queue but does not independently prescribe the eviction discipline. A stack reverses insertion order, and a generic binary tree provides no FIFO eviction contract.

Why each choice is right or wrong

A - Incorrect: LIFO would evict the most recent insertion rather than the oldest.

B - Correct: FIFO queue service matches insertion-order eviction.

C - Incorrect: A linked list is a possible representation but does not itself prescribe FIFO service.

D - Incorrect: A generic binary tree does not prescribe insertion-order eviction.

Primary note proof

Study-note section DSA27

> A dictionary or compiler symbol table can use hashing for expected constant lookup, or a balanced tree for logarithmic lookup with ordering. A bounded cache also needs eviction rules: FIFO uses a queue; LRU can combine hash lookup with a doubly linked recency list. Move a hit to the recent end and evict the least-recent end when full. No storage structure is universally best without operation and policy requirements.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

> “A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
> The two basic operations are:
> Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
> Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various

operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Question sources

Computer Science MCQ practice collection · Question 428

Question 166 | AM0429

Which data structure is commonly used for implementing undo-redo functionality in text editors?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: Stack

Undo applies the inverse of the latest action, a LIFO operation. Keeping undone actions on another stack supports redo in the reverse order of undo. Evidence: assigned PDF pages 7, 6.

Why each choice is right or wrong

A - Correct: Two stacks implement the required action reversal and restoration ordering. See assigned PDF 7, 6.

B - Incorrect: FIFO would undo actions in their original order. See assigned PDF 7, 6.

C - Incorrect: A linked list can store history but needs an added stack discipline. See assigned PDF 7, 6.

D - Incorrect: A binary tree does not intrinsically select the latest action. See assigned PDF 7, 6.

Primary note proof

Study-note section DSA12

Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 429

Question 167 | AM0430

Which data structure allows easy traversal in a Last-In-First-Out (LIFO) order?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: Stack

Easy traversal here is interpreted as repeated access/removal by the defining ADT rule. A stack pops the most recently pushed remaining element first. Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Correct: Stack expressly enforces last-in-first-out. See assigned PDF 6.

B - Incorrect: Queue expressly enforces first-in-first-out. See assigned PDF 6.

C - Incorrect: A linked list can emulate LIFO only after choosing an endpoint policy. See assigned PDF 6.

D - Incorrect: A binary tree traversal is determined by child order, not insertion recency. See assigned PDF 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

> “A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
> • Push To add an element to the stack
> • Pop To remove an element from the stock
> • Peek To look at elements in the stack without removingthem
> • LIFO Refers to the last in, first out behavior of thestack
> • FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 430

Question 168 | AM0431

Which specified representation supports constant-time insertion/deletion at either endpoint and constant-time indexed access to the logical middle item? Assume fixed capacity, no shifting or resizing, spare space for insertion, and a nonempty sequence for access.

A. Stack

B. Queue

C. Fixed-capacity circular-array deque

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: C. Fixed-capacity circular-array deque

A circular-array deque maps logical position k to a slot by adding the front index and reducing modulo capacity. That makes the middle item directly indexable. Endpoint insertions and deletions change only a slot and constant-size indices when no resize or shift occurs. A linked deque would not provide this indexing guarantee, so the array representation is explicit. The other ordinary ADT interfaces lack the combined operation set.

Why each choice is right or wrong

A - Incorrect: A stack does not expose both-end updates and middle indexing.

B - Incorrect: An ordinary queue does not expose all four endpoint operations or arbitrary middle access.

C - Correct: The fixed circular-array deque supplies endpoint updates and indexed middle access.

D - Incorrect: A generic binary tree lacks constant-time access by middle sequence position.

Primary note proof

Study-note section DSA16

> A deque permits both-end insertion/removal without guaranteeing middle access. Endpoint-linked operations are constant but middle indexing is linear. A circular-array deque may support indexed access. Dequeue names removal; deque names the structure.

Study-note section DSA15

> insertion slot is (front+size)%C.

Study-note section DSA02

> This arithmetic gives constant-time positional access in the usual RAM cost model.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 8 | lines 38-40

> Deque (pronounced as Deck): is a Double Ended Queue in which insertion and deletion can occur at either end.
> Priority Queue:- is a queue where each data has an associated key provided at insertion time.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node

pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists. The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to the user. An underflow error occurs if the queue is empty.

Question sources

Computer Science MCQ practice collection · Question 431

Question 169 | AM0432

Which data structure is commonly used for implementing the undo operation in a text editor?

A. Stack

B. Queue

C. Linked list

D. Binary tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: Stack

The latest action must be reversed first, matching LIFO and the source backtracking application. Pushing action records and popping the latest implements that order. Evidence: assigned PDF pages 6, 7.

Why each choice is right or wrong

A - Correct: Top holds the latest pending action to undo. See assigned PDF 6, 7.

B - Incorrect: FIFO selects the oldest pending action instead. See assigned PDF 6, 7.

C - Incorrect: A generic linked list needs a chosen stack discipline to implement undo. See assigned PDF 6, 7.

D - Incorrect: A binary tree alone does not impose reverse chronology. See assigned PDF 6, 7.

Primary note proof

Study-note section DSA12

Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

“A stack is an ordered list in which all insertions and deletions are made at one end, called the top”. Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Data Structure and Algorithms module | PDF page 7 | lines 2-8

• Reversing Data and Reverse a List
• Convert Decimal to Binary
• Parsing - It is a logic that breaks into independent pieces for further processing
• Backtracking
Examples:
1. Infix notation: A+(B*C) Equivalent Postfix notation ABC*+
2. Infix notation: (A+B)*C Equivalent Postfix notation AB+C*

Question sources

Computer Science MCQ practice collection · Question 432

Question 170 | AM0433

A bounded FIFO buffer must reuse freed array slots by wrapping indices modulo its fixed capacity, without shifting items. New insertions are rejected while full. Which listed term specifically names this storage representation?

A. Stack

B. Queue

C. Linked list

D. Circular buffer

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer D: D. Circular buffer

A circular buffer reuses its fixed slots by wrapping indices modulo capacity, so released positions can be used again without moving surviving items. The FIFO order is tracked logically rather than by requiring the front to remain at physical slot zero. Queue names the service ADT and can have other representations; it does not specifically identify this layout. Stack and linked list likewise do not name the stated circular-array representation.

Why each choice is right or wrong

A - Incorrect: A stack names LIFO service rather than the specified FIFO ring storage.

B - Incorrect: Queue names the FIFO ADT but does not specifically identify its storage layout.

C - Incorrect: A linked list does not name modulo-indexed circular array slots.

D - Correct: Circular buffer specifically names the stated fixed-slot wraparound representation.

Primary note proof

Study-note section DSA15

> A simple noncircular array queue may report overflow when rear reaches the final index despite released slots at the front. A circular queue reuses slots with modulo arithmetic. Use a count convention: front points to the next removal, size counts items, and insertion slot is (front+size)%C. Empty means size==0; full means size==C.

Study-note section DSA16

> Merging FIFO queues needs an explicit policy: append one sequence after another or alternate their fronts. Splitting by a predicate preserves each subgroup’s original order if each removed item is appended to its matching output queue. A fixed-capacity buffer also needs a full-policy choice: reject, block or overwrite. Capacity alone does not choose a policy.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 7 | lines 24-26

> II. Circular array implementation of Queue Operations
> To overcome the above drawback of simple array implementation a circular array can be used, in
> which freed spaces are re-used to store data.

Data Structure and Algorithms module | PDF page 7 | lines 18-23

> I. Simple Array implementation (static memory).
> A simple array structure can be used to implement enqueue & dequeue operations. Thus item will
> be added at the end of the array whereas dequeue operation removes items from the front. In

simple
array implementation, to add item it checks whether there is room at the rear or not. Therefore, even though there are free released spaces at the front ot the array the queue implementation might
generate overflow message. This is the key drawback to use simple array to implement Queue.

Question sources

Computer Science MCQ practice collection · Question 433

Question 171 | AM0434

Which ordinary array sorting algorithms have tight worst-case time Theta(n^2)? Select all that apply. Use buffered mergesort, ordinary quicksort, standard insertion sort and iterative heapsort.

A. Merge sort

B. Quick sort

C. Insertion sort

D. Heap sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B, C: B. Quick sort; C. Insertion sort

Repeated extreme partitions make ordinary quicksort accumulate quadratic work. Insertion sort on reverse input similarly accumulates quadratic prefix shifts. Buffered mergesort and iterative heapsort have n log n worst-case bounds instead, so they are not tightly quadratic even though quadratic is a valid looser upper bound. The explicit tight-bound question therefore selects quicksort and insertion alone in this primary option set.

Why each choice is right or wrong

A - Incorrect: Buffered mergesort has n log n worst growth.

B - Correct: Extreme quicksort partitions give tight quadratic worst growth.

C - Correct: Reverse-order insertion gives tight quadratic worst growth.

D - Incorrect: Iterative heapsort has an n log n worst bound.

Primary note proof

Study-note section DSA36

Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work.

Study-note section DSA34

Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Study-note section DSA37

There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA38

Building with bottom-up sinking is linear; repeated extraction gives O(n log n) worst time.

Related reading (context, not primary note proof)

Study-note section DSA32

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Variant 1 | Alternative version 1

Which ordinary array sorting algorithms have tight worst-case time Theta(n^2)? Use strict-swap bubble sort, ordinary quicksort, buffered mergesort and standard insertion sort. Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Answer A, B, D: A. Bubble sort; B. Quick sort; D. Insertion sort

Bubble sort can require quadratically many adjacent comparisons, quicksort can accumulate quadratic work through extreme pivots, and insertion sort can perform quadratic shifts on reversed input. Buffered mergesort has n log n growth instead. The variant has bubble, quick, merge and insertion in its own label order, so its complete tight-quadratic selection is A, B and D.

Why each choice is right or wrong

A - Correct: Adjacent bubble passes can require quadratic work.

B - Correct: Extreme quicksort partitions can require quadratic work.

C - Incorrect: Buffered mergesort has n log n growth.

D - Correct: Reverse input causes quadratic insertion shifts.

Primary note proof

Study-note section DSA36

> Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work.

Study-note section DSA34

> Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Study-note section DSA32

> Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Related reading (context, not primary note proof)

Study-note section DSA38

Question sources

Computer Science MCQ practice collection · Question 434

Computer Science MCQ practice collection · Question 444

Question 172 | AM0435

Which sorting algorithm uses the divide-and-conquer strategy? Use ordinary array implementations, with radix sorting specifically meaning iterative least-significant-digit passes.

A. Bubble sort

B. Selection sort

C. Merge sort

D. Radix sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: C. Merge sort

Mergesort divides the array into two halves, recursively sorts them and merges their results, so it uses divide and conquer. Ordinary bubble sort uses adjacent passes and selection sort repeatedly selects a suffix minimum. The explicitly specified LSD radix method performs stable digit passes rather than recursive subproblem decomposition. This convention removes the possible confusion with MSD radix grouping while retaining the original primary answer.

Why each choice is right or wrong

A - Incorrect: Ordinary bubble sort uses adjacent passes rather than recursive decomposition.

B - Incorrect: Ordinary selection sort repeatedly places one suffix minimum.

C - Correct: Mergesort recursively solves smaller halves and combines their results.

D - Incorrect: The specified iterative LSD form uses digit passes rather than recursive divide and conquer.

Primary note proof

Study-note section DSA-S04

> Divide and conquer reduces a problem to smaller instances of the same kind, solves the required smaller instances and obtains the original result from them. Mergesort recursively sorts two halves and merges; quicksort partitions and recursively sorts the resulting ranges. This course also classifies binary search as divide and conquer: an ordered comparison selects the only half that can contain the target. This is different from simply repeating adjacent swaps, choosing successive minima, or making stable LSD digit passes.

Study-note section DSA39

> LSD radix sorting handles fixed-width digit keys by stably sorting least-significant digit first, then moving left. Stable intermediate passes preserve the ordering already established by less-significant digits. With d digits and radix r, counting-based passes take O(d(n+r)) time and O(n+r) auxiliary storage.

Study-note section DSA34

> Begin with the first element already sorted. For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Related reading (context, not primary note proof)

Study-note section DSA36

Study-note section DSA37

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

> iv. Merge sort
> Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
> Algorithm:

1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Variant 1 | Alternative version 1

Which ordinary sorting algorithms use divide and conquer by recursively sorting smaller subproblems? Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Answer B, C: B. Quick sort; C. Merge sort

Quicksort partitions around a pivot and recursively sorts the smaller ranges. Top-down mergesort recursively sorts halves and merges their results. Both therefore apply divide and conquer. Bubble sort's adjacent passes and ordinary insertion sort's growing sorted prefix do not use this recursive subproblem decomposition. The complete answer is B and C; the question requires multiple selection.

Why each choice is right or wrong

A - Incorrect: Bubble sorting repeatedly swaps adjacent inversions instead of recursively decomposing subinstances.

B - Correct: Quicksort partitions around a pivot and recursively sorts the smaller ranges.

C - Correct: Merge sort recursively sorts halves and combines their sorted results.

D - Incorrect: Ordinary insertion sort extends a sorted prefix instead of applying recursive divide and conquer.

Primary note proof

Study-note section DSA-S04

> Divide and conquer reduces a problem to smaller instances of the same kind, solves the required smaller instances and obtains the original result from them. Mergesort recursively sorts two halves and merges; quicksort partitions and recursively sorts the resulting ranges. This course also classifies binary search as divide and conquer: an ordered comparison selects the only half that can contain the target. This is different from simply repeating adjacent swaps, choosing successive minima, or making stable LSD digit passes.

Study-note section DSA36

> Choose a pivot, partition, then recursively sort the pieces. Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work. Expected n log n needs an input/random-pivot model. Random and median-of-three choices have different tradeoffs; neither is universally best.

Study-note section DSA37

> Split into halves, sort them, then merge by advancing the smaller current front. With left-first tie handling, equal keys retain order. Ordinary array merging needs a linear auxiliary buffer. There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA34

> Begin with the first element already sorted. For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Related reading (context, not primary note proof)

Study-note section DSA39

Question sources

Computer Science MCQ practice collection · Question 435

Computer Science MCQ practice collection · Question 445

Question 173 | AM0436

Which specified sorting implementations preserve the relative order of equal full keys? Use ordinary in-place quicksort, ordinary heapsort, LSD radix with stable digit passes, and mergesort with left-first ties. Select all that apply.

A. Quick sort

B. Heap sort

C. Radix sort

D. Merge sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C, D: C. Radix sort; D. Merge sort

Stable LSD radix passes preserve the ordering already established by less significant digits. Equal full keys remain tied on every pass, so their relative order remains unchanged. Mergesort with left-first ties also preserves equal-key order while combining runs. Ordinary in-place quicksort and heapsort do not guarantee that stability because distant exchanges can reorder equal records. Hence both radix and merge qualify under the specified implementations.

Why each choice is right or wrong

A - Incorrect: Ordinary quicksort does not guarantee equal-key stability.

B - Incorrect: Long heap exchanges can reorder equal records.

C - Correct: Stable digit passes preserve the order of equal full keys.

D - Correct: Left-first merge ties preserve equal-key order.

Primary note proof

Study-note section DSA39

> Stable intermediate passes preserve the ordering already established by less-significant digits.

Study-note section DSA37

> With left-first tie handling, equal keys retain order.

Study-note section DSA38

> The method is generally unstable because long root-to-end exchanges can reorder equal keys.

Study-note section DSA31

stack usually log n average, n worst; unstable

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Computer Science MCQ practice collection · Question 436

Question 174 | AM0437

Which sorting algorithm uses a pivot element to partition the array into two subarrays?

A. Bubble sort

B. Selection sort

C. Quick sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Quick sort

The module quicksort explicitly selects a pivot and partitions values below/above it before recursion. Evidence: assigned PDF pages 11, 4, 3.

Why each choice is right or wrong

A - Incorrect: Bubble uses adjacent comparisons and swaps. See assigned PDF 11, 4, 3, 3.

B - Incorrect: Selection chooses the minimum for the next output slot. See assigned PDF 11, 4, 3, 3.

C - Correct: Pivot partitioning is the stated quicksort procedure. See assigned PDF 11, 4, 3, 3.

D - Incorrect: Insertion puts each item into a sorted prefix. See assigned PDF 11, 4, 3, 3.

Primary note proof

Study-note section DSA36

> Choose a pivot, partition, then recursively sort the pieces. Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work. Expected n log n needs an input/random-pivot model. Random and median-of-three choices have different tradeoffs; neither is universally best.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

> ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
> Algorithm:
> 1. Choose a pivot value (mostly the first element is taken as the pivot value)
> 2. Position the pivot element and partition the list so that:
> • the left part has items less than or equal to the pivot value
> • the right part has items greater than or equal to the pivot value
> 3. Recursively sort the left part
> 4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 437

Question 175 | AM0439

Which searching algorithm is based on comparing the target value with elements at midpoints of the array?

A. Linear search

B. Binary search

C. Interpolation search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B: Binary search

DSA29 computes the midpoint of the current ordered interval, compares it with the target, and discards the half that cannot contain the target. That is exactly the midpoint-based procedure in the stem. Linear search scans successive elements, interpolation estimates a position from values, and DFS follows graph branches rather than ordered-array midpoints.

Why each choice is right or wrong

A - Incorrect: Linear search compares successive elements rather than repeatedly choosing the midpoint of a shrinking ordered interval.

B - Correct: Binary search explicitly compares the target with the current interval midpoint and keeps one half.

C - Incorrect: Interpolation search estimates a probe from numeric values and their distribution, rather than always using the midpoint.

D - Incorrect: Depth-first search explores graph neighbors and branches, not ordered-array midpoint intervals.

Primary note proof

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Study-note section DSA30

> Interpolation search estimates the index from numeric values: lo+(hi-lo)×(key-A[lo])/(A[hi]-A[lo]), rounded to a valid index. Guard equal endpoints and out-of-range targets. On sorted, suitably uniform data its expected probes can be O(log log n); pathological distributions can require linear time. Binary search gives a logarithmic worst bound without that distribution assumption.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first
> element
> • Time is proportional to the size of input
> (n) and we call this time complexity
> O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science MCQ practice collection · Question 439

Question 176 | AM0440

Which ordinary array-search algorithms require sorted input? For interpolation search also assume comparable numeric values and appropriate endpoint guards. Select all that apply.

A. Linear search

B. Binary search

C. Interpolation search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B, C: B. Binary search; C. Interpolation search

Binary search relies on order to discard half the remaining candidate interval after a comparison. Interpolation search also relies on sorted numeric values when estimating a candidate position from endpoint values. Both therefore require sorted input, although their further assumptions differ. Linear search simply scans and permits unsorted data. DFS follows graph adjacency rather than relying on a sorted array-search interval.

Why each choice is right or wrong

A - Incorrect: Linear scanning permits unsorted data.

B - Correct: Binary interval elimination needs sorted order.

C - Correct: Interpolation index estimation also needs sorted input.

D - Incorrect: DFS follows graph connections and has no sorted-array requirement.

Primary note proof

Study-note section DSA29

On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Study-note section DSA30

Interpolation search estimates the index from numeric values: lo+(hi-lo)×(key-A[lo])/(A[hi]-A[lo]), rounded to a valid index. Guard equal endpoints and out-of-range targets. On sorted, suitably uniform data its expected probes can be O(log log n); pathological distributions can require linear time. Binary search gives a logarithmic worst bound without that distribution assumption.

Study-note section DSA28

Unsorted data is allowed; sorted data is allowed too.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science MCQ practice collection · Question 440

Question 177 | AM0441

Which searching algorithm works by comparing the target value with elements from the beginning of the array until a match is found?

A. Linear search

B. Binary search

C. Interpolation search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: A. Linear search

Linear search starts at the beginning and compares successive elements until a match or exhaustion. Binary search instead chooses a midpoint of an ordered interval, and interpolation search estimates a position from endpoint key values. Depth-first search follows an unvisited graph branch before returning. These positive procedural differences classify the alternatives without any appeal to an old module's omissions, so A remains the definite matching choice.

Why each choice is right or wrong

A - Correct: Linear search compares successive elements from the beginning.

B - Incorrect: Binary search probes a midpoint rather than scanning every successive position.

C - Incorrect: Interpolation search estimates a probe position from endpoint values.

D - Incorrect: DFS follows graph adjacency and unfinished branches rather than a beginning-to-end array scan.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted.

Study-note section DSA29

> Compute mid=lo+(hi-lo)//2.

Study-note section DSA30

> Interpolation search estimates the index from numeric values: lo+(hi-lo)×(key-A[lo])/(A[hi]-A[lo]), rounded to a valid index.

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science MCQ practice collection · Question 441

Question 178 | AM0442

Which searching algorithm is more efficient for uniformly distributed data in a sorted array?

A. Linear search

B. Binary search

C. Interpolation search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Interpolation search under the stated uniform-data model.

DSA30 gives interpolation search expected O(log log n) probes on sorted, suitably uniform numeric data. Binary search has logarithmic worst-case probes without that distribution assumption, while linear search scans. The stem supplies the distribution and sortedness conditions that motivate interpolation. The expected advantage does not promise faster finite runs or logarithmic worst-case behavior on arbitrary distributions.

Why each choice is right or wrong

A - Incorrect: Linear search still examines successive elements and does not exploit uniform numeric distribution for sublinear expected probing.

B - Incorrect: Binary search gives logarithmic worst-case probing, but does not exploit uniform values for the stated expected advantage.

C - Correct: Interpolation estimates positions from uniformly distributed sorted numeric values, supporting the note's expected O(log log n) probes.

D - Incorrect: DFS explores graph or tree branches and is not the distribution-based ordered-array search described here.

Primary note proof

Study-note section DSA30

> Interpolation search estimates the index from numeric values: lo+(hi-lo)×(key-A[lo])/(A[hi]-A[lo]), rounded to a valid index. Guard equal endpoints and out-of-range targets. On sorted, suitably uniform data its expected probes can be O(log log n); pathological distributions can require linear time. Binary search gives a logarithmic worst bound without that distribution assumption.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first element
> • Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science MCQ practice collection · Question 442

Question 179 | AM0443

Which named algorithms explore a graph or tree by following adjacency links to unvisited vertices, using branch-first or distance-layer-first traversal? Select all that apply.

A. Linear search

B. Binary search

C. Depth-first search

D. Breadth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C, D: C. Depth-first search; D. Breadth-first search

DFS explores an unvisited branch before returning to later neighbors, using recursion or a stack. BFS instead processes a FIFO frontier so successive distance layers are explored in order. Both follow adjacency relationships and apply to trees as well as general graphs. Linear sequence scanning and binary interval search are different algorithms and do not implement these two adjacency-frontier traversal disciplines.

Why each choice is right or wrong

A - Incorrect: Linear sequence search does not implement the specified adjacency-frontier traversal.

B - Incorrect: Binary search eliminates ordered interval halves rather than following arbitrary adjacency.

C - Correct: DFS is branch-first adjacency traversal.

D - Correct: BFS is distance-layer-first adjacency traversal.

Primary note proof

Study-note section DSA41

> Mark the source when enqueuing it. Repeatedly dequeue a vertex, examine its neighbours, and enqueue each unvisited neighbour while marking it. Store predecessor[v] when discovering v if a path must be reconstructed. FIFO ensures vertices at distance d are explored before later distance layers.

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 40-44

III. Tree Traversing:
Binary search tree can be traversed in three ways.
a. Pre order traversal - traversing binary tree in the order of parent, left and right.
b. Inorder traversal - traversing binary tree in the order of left, parent and right.
c. Postorder traversal - traversing binary tree in the order of left, right and parent.

Question sources

Computer Science MCQ practice collection · Question 443

Question 180 | AM0446

Under a uniform random permutation of distinct keys, which ordinary sorting algorithms have tight average time Theta(n log n)? Use fixed-position-pivot quicksort, buffered mergesort without merge-skipping, and standard bubble/insertion implementations. Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B, C: B. Quick sort; C. Merge sort

The random-order model supplies the average-case assumption needed for ordinary quicksort's expected n log n work. Buffered mergesort performs n log n work across input orders, so its average has the same order. Bubble and insertion have quadratic ordinary average behavior under this random permutation model. Tight bounds and the stated distribution make quicksort and mergesort the complete selection, rather than leaving average cost unspecified. The DSA31 comparison table supplies the average-case entries used in this comparison.

Why each choice is right or wrong

A - Incorrect: Ordinary bubble sorting has quadratic random-order average work.

B - Correct: The stated random-input model gives quicksort expected n log n work.

C - Correct: Buffered mergesort has n log n work in every input order.

D - Incorrect: Ordinary insertion has quadratic random-order average work.

Primary note proof

Study-note section DSA31

n log n / n log n / n^2

Study-note section DSA31

n / n^2 / n^2

Study-note section DSA37

There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd

item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 446

Question 181 | AM0447

Which sorting algorithm works by repeatedly selecting the smallest element from the unsorted portion and placing it at the beginning?

A. Bubble sort

B. Quick sort

C. Selection sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Selection sort

The source selection sort scans for the smallest remaining element and swaps it with the current first unsorted position. Evidence: assigned PDF pages 3, 4, 11.

Why each choice is right or wrong

A - Incorrect: Bubble pushes elements through adjacent swaps. See assigned PDF 3, 4, 11, 3.

B - Incorrect: Quicksort partitions by a pivot, not necessarily the minimum. See assigned PDF 3, 4, 11, 3.

C - Correct: Exactly the minimum-selection and placement procedure given. See assigned PDF 3, 4, 11, 3.

D - Incorrect: Insertion places the next encountered element into a sorted prefix. See assigned PDF 3, 4, 11, 3.

Primary note proof

Study-note section DSA33

At position i, scan the remaining suffix for its minimum, then exchange it with A[i]. There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order. At most n-1

exchanges of distinct positions are needed. A loop that also swaps the last item with itself can count n swap calls.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 44-48

B. Selection Sort
Loop through the list/array from i=0 to n-1 and select the smallest element in the list and swap this value with value at position i. Repeat this step for the rest of the items.
• How many comparisons? => O(n2
) and How many swaps? => O(n)

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in
the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 447

Question 182 | AM0448

Which ordinary array sorting algorithms are comparison sorts that use constant extra element storage inside the original array? Count recursive call stacks separately from element storage; use buffered mergesort for C. Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A, B, D: A. Bubble sort; B. Quick sort; D. Insertion sort

Bubble sort swaps adjacent keys and insertion sort shifts a prefix while retaining one saved key. Quicksort partitions within the same array using constant extra element storage. Its recursive frames are counted separately under the explicitly stated convention. Buffered mergesort instead allocates a linear merge buffer. Therefore bubble, quick and insertion meet the element-storage definition, while a total-auxiliary-space question would need to account for quicksort's stack as well.

Why each choice is right or wrong

A - Correct: Bubble compares and swaps within the original array.

B - Correct: Quicksort partitions in the array; its recursive stack is separately counted.

C - Incorrect: Buffered mergesort needs linear extra element storage.

D - Correct: Insertion compares and shifts within the original array with one saved key.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right.

Study-note section DSA34

> For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Study-note section DSA36

> Quicksort normally rearranges in the same array but recursive storage can be linear in the worst case.

Study-note section DSA37

Ordinary array merging needs a linear auxiliary buffer.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.

• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 448

Question 183 | AM0449

Which searching algorithm requires the input array to be sorted?

A. Linear search

B. Binary search

C. Hash search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B: Binary search

DSA29 defines binary search over an ascending random-access sequence and reverses comparisons for descending data. Sortedness makes the half-elimination step sound. Linear scanning works without order; hashing needs a built table rather than sorted positions; DFS follows adjacency. Only Binary search among this option set requires a sorted input sequence for its ordinary operation.

Why each choice is right or wrong

A - Incorrect: Linear search explicitly permits unsorted data and compares items one after another.

B - Correct: Binary search relies on ordered values to determine which half can still contain the target.

C - Incorrect: Hash lookup needs a hash-table organization, not values arranged in sorted-array order.

D - Incorrect: Depth-first search follows graph or tree adjacency and does not require a sorted input array.

Primary note proof

Study-note section DSA29

On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi.

Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Study-note section DSA25

Direct addressing reserves a position for every possible key in a finite universe U, costing Theta(|U|) slots. A hash table uses h(key) in 0..m-1 and stores only actual entries plus bucket capacity. With h(k)=k mod 6, key 36 has home address 0; collisions may move its final storage position.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science MCQ practice collection · Question 449

Question 184 | AM0450

Which searching algorithm has a worst-case time complexity of O(log n)?

A. Linear search

B. Binary search

C. Hash search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B: Binary search has logarithmic worst-case time.

DSA29 bounds binary-search probes by floor($\log_2 n$)+1 on nonempty random-access input. This supplies logarithmic worst-case time under constant-cost comparisons. Linear search may scan all items; adverse hash collisions may require linear work; DFS can inspect all vertices and edges. Their ordinary worst cases do not share binary search's logarithmic bound.

Why each choice is right or wrong

A - Incorrect: Linear search can inspect every item when the target is absent or last, giving linear worst-case time.

B - Correct: Each comparison discards about half the ordered interval, giving logarithmic worst-case probes.

C - Incorrect: Ordinary hash lookup can degrade to linear time under adverse collisions, despite expected constant-time performance.

D - Incorrect: DFS may visit all graph vertices and edges, rather than guarantee logarithmic worst-case work.

Primary note proof

Study-note section DSA29

> Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Study-note section DSA26

> Separate chaining stores colliding entries in a bucket list. Load alpha=n/m is average chain length; expected operations are O(1+alpha) under suitable distribution assumptions, with O(n) worst cases. Open addressing keeps entries inside the array and probes until a key or an available slot is found.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science MCQ practice collection · Question 450

Question 185 | AM0451

Which search compares a target with consecutive sequence elements at a fixed stride of one, stopping at the first match or after exhausting the sequence? No hashing or other preprocessing is used.

A. Linear search

B. Binary search

C. Hash search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: A. Linear search

The stated unit stride means the algorithm inspects successive elements in order. It stops early when a comparison matches or reports failure after the sequence is exhausted, which is linear search. Binary search uses a shrinking midpoint interval rather than consecutive positions. Hash search starts from computed addresses, and DFS follows graph adjacency. Specifying stride one distinguishes ordinary scanning from larger-step jump-search interpretations.

Why each choice is right or wrong

A - Correct: Linear search compares successive elements until a match or exhaustion.

B - Incorrect: Binary search chooses midpoints rather than every consecutive element.

C - Incorrect: Hash search uses computed addresses rather than the stated scan.

D - Incorrect: DFS follows graph adjacency rather than consecutive sequence positions.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Study-note section DSA25

> Direct addressing reserves a position for every possible key in a finite universe U, costing Theta(|U|) slots. A hash table uses h(key) in 0..m-1 and stores only actual entries plus bucket capacity. With h(k)=k mod 6, key 36 has home address 0; collisions may move its final storage position.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

> Linear/Sequential searching
> • Works with unsorted List
> • Loop through the list starting at the first element
> • Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
> • The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science MCQ practice collection · Question 451

Question 186 | AM0452

Which searching algorithm is suitable for unsorted arrays or lists?

A. Linear search

B. Binary search

C. Hash search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A: Linear search directly searches an unsorted sequence.

DSA28 explicitly allows unsorted data for linear search: compare successive elements until the target is found or the sequence ends. Binary search requires sortedness. Hash lookup can be useful after building a table, but it does not directly search the supplied ordinary list without that preprocessing. DFS presumes graph or tree adjacency rather than a simple sequence query.

Why each choice is right or wrong

A - Correct: Linear search can inspect an unsorted array or list directly without sorting or building another structure.

B - Incorrect: Binary search needs ordering to safely eliminate half the remaining candidate positions.

C - Incorrect: Hash search needs a constructed hash table; the stem supplies an ordinary unsorted array or list.

D - Incorrect: DFS explores graph or tree adjacency and is not the direct unsorted-sequence scan described here.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Study-note section DSA25

Direct addressing reserves a position for every possible key in a finite universe U, costing Theta(|U|) slots. A hash table uses h(key) in 0..m-1 and stores only actual entries plus bucket capacity. With h(k)=k mod 6, key 36 has home address 0; collisions may move its final storage position.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science MCQ practice collection · Question 452

Question 187 | AM0453

Which searching algorithm is commonly used for traversing graphs or trees?

A. Linear search

B. Binary search

C. Hash search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer D: Depth-first search

DSA42 describes marking a vertex and recursively visiting each unmarked neighbor before resuming later neighbors. That procedure is standard depth-first traversal and applies to graphs and trees. Linear search scans a sequence, binary search uses an ordered interval, and hash search queries table addresses. None of those alternatives names the adjacency-following traversal in this option set.

Why each choice is right or wrong

A - Incorrect: Linear search scans sequence elements and is not the named adjacency-following graph/tree traversal.

B - Incorrect: Binary search reduces an ordered interval rather than visiting arbitrary graph or tree connections.

C - Incorrect: Hash search probes key-derived table addresses rather than traversing graph or tree edges.

D - Correct: DFS follows unvisited neighbors recursively or through an explicit stack, providing a standard traversal.

Primary note proof

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 40-44

> III. Tree Traversing:
> Binary search tree can be traversed in three ways.
> a. Pre order traversal - traversing binary tree in the order of parent, left and right.
> b. Inorder traversal - traversing binary tree in the order of left, parent and right.
> c. Postorder traversal - traversing binary tree in the order of left, right and parent.

Question sources

Computer Science MCQ practice collection · Question 453

Question 188 | AM0455

Which specified sorting implementations are stable? Use strict adjacent swaps for bubble, ordinary in-place quicksort, left-first merge ties, and insertion shifts of strictly larger keys only. Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A, C, D: A. Bubble sort; C. Merge sort; D. Insertion sort

Bubble sort's strict adjacent swaps do not exchange equal keys. Mergesort preserves ties by taking an equal key from the left run first. Insertion sort shifts only strictly larger keys, leaving equal earlier keys ahead of the inserted one. Those three mechanisms preserve relative equal-key order. Ordinary in-place quicksort does not guarantee it because partition exchanges can move equal records past one another.

Why each choice is right or wrong

A - Correct: Strict adjacent swaps preserve ties.

B - Incorrect: Ordinary quicksort does not guarantee equal-key order.

C - Correct: Left-first equal-key merging preserves ties.

D - Correct: Shifting only larger keys leaves equal earlier keys in place.

Primary note proof

Study-note section DSA32

> Strict swaps preserve equal-key order.

Study-note section DSA34

> Shift only strictly larger keys for stability.

Study-note section DSA37

> With left-first tie handling, equal keys retain order.

Study-note section DSA31

> stack usually log n average, n worst; unstable

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 455

Question 189 | AM0456

Which sorting algorithm is commonly used for small input sizes or partially sorted arrays?

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer D: Insertion sort

DSA34 explicitly favors insertion for small arrays or few inversions. Each inversion corresponds to shifting work, so nearly sorted data requires few shifts; sorted input is linear. This low-setup incremental algorithm is the conventional match among the choices. The other algorithms can sort these inputs, but their mere applicability does not identify the specific use case emphasized in the note.

Why each choice is right or wrong

A - Incorrect: Bubble can exploit early stopping but is not the note's specific small-array/few-inversions recommendation.

B - Incorrect: Quicksort incurs partition and recursive setup and is not the cited small-subarray method.

C - Incorrect: Ordinary merge sorting uses its merge levels and buffer even on nearly sorted input.

D - Correct: Insertion has low setup and few shifts on nearly sorted or small arrays.

Primary note proof

Study-note section DSA34

> Few inversions mean few shifts, favouring insertion for nearly sorted or small arrays. Shift only strictly larger keys for stability. A linked implementation saves array shifts but still must locate each insertion position.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 34-43

> A. Insertion Sort
> The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put

them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Computer Science MCQ practice collection · Question 456

Question 190 | AM0457

Which searching algorithm divides the search space into three parts at each step?

A. Linear search

B. Binary search

C. Ternary search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Ternary search

DSA30 explicitly describes sorted-array ternary search as comparing two split points and retaining the relevant third. The three candidate regions give the technique its name. Binary search uses one midpoint and two regions, linear search scans successively, and DFS explores graph branches. Thus Ternary search uniquely matches the described three-part split.

Why each choice is right or wrong

A - Incorrect: Linear search advances through successive items and does not divide the remaining interval into thirds.

B - Incorrect: Binary search uses one midpoint and retains one of two halves, not one of three regions.

C - Correct: Ternary search uses two split points to divide the ordered search interval into three candidate parts.

D - Incorrect: DFS explores graph or tree branches rather than repeatedly trisecting an ordered search interval.

Primary note proof

Study-note section DSA30

> Sorted-array ternary search compares two split points and keeps the relevant third, still logarithmic. Jump search probes block boundaries then scans within a block. Regular intervals could mean unit-step scanning or larger jumps. Interpolation also needs sortedness, so prerequisite questions can overlap binary search.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

> Binary Searching
> • Works on ordered/sorted list
> • Uses divide and conquer method: it locates
> mid-point and identify which have the target

belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science MCQ practice collection · Question 457

Question 191 | AM0458

Which ordinary array sorting implementations use O(1) total auxiliary space in both best and average cases? Count live recursive frames; use recursive quicksort without tail-call elimination and buffered mergesort. Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A, D: A. Bubble sort; D. Insertion sort

The DSA31 table gives bubble and insertion constant extra storage across input cases. Their iterative forms retain only a fixed number of indices and temporaries, so total auxiliary space stays constant. The same table gives ordinary recursive quicksort a logarithmic average stack, and buffered mergesort needs a linear auxiliary array. Counting all live frames and buffers therefore leaves exactly bubble and insertion as the qualifying choices.

Why each choice is right or wrong

A - Correct: Bubble uses fixed temporary storage across cases.

B - Incorrect: The recursive quicksort stack is not constant in its ordinary average case.

C - Incorrect: Buffered mergesort uses linear auxiliary storage.

D - Correct: Insertion needs a saved key and fixed-size indices.

Primary note proof

Study-note section DSA32

For ascending order, compare each adjacent pair and swap only when left>right.

Study-note section DSA34

For each remaining item, save its value, shift larger preceding values right, and insert it into the gap.

Study-note section DSA31

stack usually log n average, n worst; unstable

Study-note section DSA37

Ordinary array merging needs a linear auxiliary buffer.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put

them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 458

Question 192 | AM0459

Which searching algorithm is based on the principle of divide-and-conquer?

A. Linear search

B. Binary search

C. Hash search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B: Binary search

DSA29 compares the middle value with the target and continues only in the half that can contain it. This solves a smaller instance of the same ordered-search problem and is the divide-and-conquer reduction intended here. Linear scanning removes one candidate at a time, hashing computes addresses, and DFS follows adjacency without this ordered half-elimination rule.

Why each choice is right or wrong

A - Incorrect: Linear search scans successive candidates rather than repeatedly solving a half-size ordered-search instance.

B - Correct: Binary search divides the ordered interval and continues in the only half that can contain the target.

C - Incorrect: Hash search derives an address and resolves collisions rather than recursively dividing an ordered problem.

D - Incorrect: DFS follows adjacency branches and does not inherently apply the ordered half-elimination strategy described here.

Primary note proof

Study-note section DSA-S04

Divide and conquer reduces a problem to smaller instances of the same kind, solves the required smaller instances and obtains the original result from them. Mergesort recursively sorts two halves and merges; quicksort partitions and recursively sorts the resulting ranges. This course also classifies binary search as divide and conquer: an ordered comparison selects the only half that can contain the target. This is different from simply repeating adjacent swaps, choosing successive minima, or making stable LSD digit passes.

Study-note section DSA29

On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Question sources

Computer Science MCQ practice collection · Question 459

Question 193 | AM0460

Which sorting algorithm has the worst-case time complexity of O(n log n)?

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Merge sort

Ordinary merge combines n items over log n levels, giving n log n even in the worst case. The other offered algorithms have quadratic adverse-input work. Evidence: assigned PDF pages 4, 11, 3.

Why each choice is right or wrong

A - Incorrect: Bubble can perform quadratic passes/exchanges. See assigned PDF 4, 11, 11, 3.

B - Incorrect: Quicksort is explicitly quadratic in the worst case. See assigned PDF 4, 11, 11, 3.

C - Correct: Merge is explicitly n log n. See assigned PDF 4, 11, 11, 3.

D - Incorrect: Reverse-order insertion requires quadratic movement. See assigned PDF 4, 11, 11, 3.

Primary note proof

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Study-note section DSA36

> Balanced partitions give n log n work; repeated extreme pivots sum n+(n-1)+... to quadratic work.

Study-note section DSA34

> Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Study-note section DSA32

> Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 460

Question 194 | AM0461

Which sorting algorithm is efficient for sorting data in external memory (hard disks)?

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Merge sort using buffered external runs.

DSA37 describes external sorting by building sorted runs and merging buffered disk streams to reduce I/O. Merge sort naturally supports that strategy even when the whole dataset does not fit RAM. Bubble and insertion sorting repeatedly move local data, while ordinary quicksort assumes partition access rather than the external sequential-run pattern specified here.

Why each choice is right or wrong

A - Incorrect: Bubble sort's repeated adjacent passes do not provide the buffered sorted-run strategy used for efficient external sorting.

B - Incorrect: Ordinary quicksort partitioning is not the sequential buffered-run merging strategy identified for external data here.

C - Correct: Merge sorting combines buffered sorted runs sequentially and is directly suited to disk-backed external sorting.

D - Incorrect: Insertion sort's repeated local insertion work does not provide the note's efficient external-run merging mechanism.

Primary note proof

Study-note section DSA37

> Top-down mergesort recurses; bottom-up merges runs of widths 1,2,4,... iteratively. In-place merging variants exist with different tradeoffs; linear merge names an operation, not a universally standardized sort variant. Internal sorting fits the working dataset in memory. External sorting builds sorted runs and merges buffered streams from disk, aiming to reduce I/O. Reverse order alone does not characterize maximum merge comparisons.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 38-44

> iv. Merge sort
> Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
> Algorithm:

1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Computer Science MCQ practice collection · Question 461

Question 195 | AM0462

A large dataset has an equality-key index implemented by a hash table with suitable key distribution and bounded load. No ordered or range queries are needed. Which listed search uses this index for expected constant-time lookup?

A. Linear search

B. Binary search

C. Hash search

D. Depth-first search

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: C. Hash search

Hash lookup computes a bucket or probe start from the key and uses the existing index to locate an equality match. With suitable distribution and bounded load, the note gives expected constant lookup. The important assumptions are the index and workload, rather than dataset size alone. Linear scanning ignores the index, binary search needs an ordered interval, and DFS follows graph relationships instead of this equality-key access structure.

Why each choice is right or wrong

A - Incorrect: A scan can inspect linearly many records and does not use the hash index.

B - Incorrect: Binary search uses an ordered interval rather than this index.

C - Correct: Hash search uses the stated index with expected constant lookup.

D - Incorrect: DFS follows graph connections rather than the equality-key index.

Primary note proof

Study-note section DSA26

Separate chaining stores colliding entries in a bucket list. Load alpha=n/m is average chain length; expected operations are O(1+alpha) under suitable distribution assumptions, with O(n) worst cases. Open addressing keeps entries inside the array and probes until a key or an available slot is found.

Study-note section DSA27

A dictionary or compiler symbol table can use hashing for expected constant lookup, or a balanced tree for logarithmic lookup with ordering. A bounded cache also needs eviction rules: FIFO uses a queue; LRU can combine hash lookup with a doubly linked recency list. Move a hit to the recent end and evict the least-recent end when full. No storage structure is universally best without operation and policy requirements.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 3 | lines 23-31

Binary Searching
• Works on ordered/sorted list
• Uses divide and conquer method: it locates mid-point and identify which have the target belongs to since the list is sorted. It performs this recursively till the item found or the list is empty(target is not in the list
• The computational time for this algorithm is proportional to log2 n.

Question sources

Computer Science MCQ practice collection · Question 462

Question 196 | AM0463

For a uniform random permutation of distinct keys, which listed ordinary algorithms have the smallest tight average-case growth order? Use fixed-position-pivot quicksort, buffered mergesort without merge-skipping, and standard bubble/insertion. Select all algorithms tied for the smallest order.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Insertion sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer B, C: B. Quick sort; C. Merge sort

Under the stated random-input model, ordinary quicksort has expected n log n work. Buffered mergesort performs n log n work in every order, so its average shares that order. Bubble and insertion have quadratic average work on random permutations. Quicksort and mergesort therefore tie for the smallest asymptotic average order among these options; the question does not ask for a measured running-time ranking between the tied methods. The DSA31 comparison table supplies the average-case entries used in this comparison.

Why each choice is right or wrong

A - Incorrect: Quadratic average growth is larger than n log n.

B - Correct: Expected n log n ties for the smallest listed average growth.

C - Correct: n log n across orders ties for the smallest listed average growth.

D - Incorrect: Quadratic average growth is larger than n log n.

Primary note proof

Study-note section DSA31

> n log n / n log n / n^2

Study-note section DSA31

> n / n^2 / n^2

Study-note section DSA37

> There are logarithmically many levels with linear merge work per level, giving n log n time even on sorted input unless an optimization skips merges.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 14-22

> ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
> Algorithm:
> 1. Choose a pivot value (mostly the first element is taken as the pivot value)
> 2. Position the pivot element and partition the list so that:
> • the left part has items less than or equal to the pivot value
> • the right part has items greater than or equal to the pivot value
> 3. Recursively sort the left part
> 4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Data Structure and Algorithms module | PDF page 3 | lines 34-43

A. Insertion Sort
The most instinctive type of sorting algorithm; it finds the location for an element and move all others up, and insert the element to its right position. It starts with the left most two items and put
them in order. Then add 3rd
item to the relatively sorted portion and find its correct location
(relative to each other) swapping adjacent elements if they are out of order. In such it sorts all item
in the list.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 463

Question 197 | AM0464

Which data structure is best suited for implementing a dictionary with fast lookup and insertion?

A. Array

B. Linked list

C. Hash table

D. Stack

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: Hash table for expected fast dictionary lookup and insertion.

DSA27 explicitly names hashing as a dictionary implementation with expected constant lookup. DSA26 qualifies this by a suitable distribution and controlled load; insertion is likewise expected fast under those conditions. Arrays and linked lists lack the same ordinary key-lookup performance, while a stack specifies LIFO access. This is an expected-performance recommendation, not a universal worst-case guarantee.

Why each choice is right or wrong

A - Incorrect: An ordinary array lacks key-derived lookup and may require scanning or maintaining a sorted order with costly insertions.

B - Incorrect: An ordinary linked list needs traversal to locate an arbitrary dictionary key, giving linear worst-case lookup.

C - Correct: A well-sized hash table offers expected fast key lookup and insertion under suitable hashing and load assumptions.

D - Incorrect: A stack exposes LIFO top operations and does not directly provide efficient arbitrary-key dictionary lookup.

Primary note proof

Study-note section DSA26

> Separate chaining stores colliding entries in a bucket list. Load alpha=n/m is average chain length; expected operations are O(1+alpha) under suitable distribution assumptions, with O(n) worst cases. Open addressing keeps entries inside the array and probes until a key or an available slot is found.

Study-note section DSA27

> A dictionary or compiler symbol table can use hashing for expected constant lookup, or a balanced tree for logarithmic lookup with ordering. A bounded cache also needs eviction rules: FIFO uses a queue; LRU can combine hash lookup with a doubly linked recency list. Move a hit to the recent end and evict the least-recent end when full. No storage structure is universally best without operation and policy requirements.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.

Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 6 | lines 15-22

"A stack is an ordered list in which all insertions and deletions are made at one end, called the top". Stacks are sometimes referred to as Last in First out (LIFO) lists. Stacks have some useful terminology associated with them:
• Push To add an element to the stack
• Pop To remove an element from the stock
• Peek To look at elements in the stack without removingthem
• LIFO Refers to the last in, first out behavior of thestack
• FILO (First in Last Out) : equivalent to LIFO

Question sources

Computer Science MCQ practice collection · Question 464

Question 198 | AM0465

Which listed algorithm guarantees shortest paths from one source to all reachable vertices for arbitrary graphs with nonnegative edge weights that may differ? Compare the standard forms of the listed algorithms.

A. Depth-first search

B. Breadth-first search

C. Dijkstra's algorithm

D. Prim's algorithm

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer C: C. Dijkstra's algorithm

Standard Dijkstra repeatedly finalizes the unsettled vertex with the smallest tentative distance and relaxes its outgoing edges. Nonnegative edge weights justify this shortest-path procedure even when different edges have different costs. BFS minimizes edge count and is sufficient when edge costs are equal, which is weaker than the stated guarantee. DFS supplies reachability paths, while Prim minimizes total spanning-tree cost rather than source-to-vertex route lengths.

Why each choice is right or wrong

A - Incorrect: A DFS path need not have minimum total weight.

B - Incorrect: BFS minimizes edge count and does not guarantee minimum weight for arbitrary differing costs.

C - Correct: Dijkstra guarantees source shortest paths under nonnegative weights.

D - Incorrect: Prim solves the spanning-tree objective rather than source shortest paths.

Primary note proof

Study-note section DSA43

> For nonnegative weights, standard Dijkstra initializes source distance zero and others infinity, repeatedly finalizes the unsettled vertex with smallest distance, and relaxes each outgoing edge: dist[v]=min(dist[v],dist[u]+w(u,v)). A min-priority queue supports the next choice. BFS is enough when every edge has equal cost; negative edges require another justified method, such as Bellman-Ford.

Study-note section DSA41

> BFS finds minimum-edge paths in unweighted graphs, not arbitrary minimum-weight paths.

Question sources

Computer Science MCQ practice collection · Question 465

Question 199 | AM0466

Which listed standard representations directly implement stack push/pop at an array top or linked-chain head, without simulating LIFO through another ADT? Select all that apply.

A. Array

B. Linked list

C. Queue

D. Tree

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A, B: A. Array; B. Linked list

The array representation maintains a top index and changes it when pushing or popping. The linked representation inserts and removes at the chain head, so the latest node is also the next one removed. Both directly implement LIFO with ordinary constant endpoint operations. A queue has FIFO service, while a generic tree has no stack-top contract; emulations through those ADTs are outside this representation question.

Why each choice is right or wrong

A - Correct: An array with a top index directly implements stack operations.

B - Correct: A linked chain with head insertion/removal directly implements stack operations.

C - Incorrect: An ordinary FIFO queue is not a direct stack representation.

D - Incorrect: A generic tree does not supply the specified stack-top representation.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA01

> Arrays or linked nodes can implement the same LIFO stack.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

> I. Array implementation (static memory).
> Push(n): Adding item at top
> top is the current top of stack and n is its
> maximum size

```
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;
```

Data Structure and Algorithms module | PDF page 6 | lines 43-45

II. Linked list (dynamic memory)
The linked list implementation of stack is adding items at the end of the chain (Push item) and deleting the end node of the list (Pop item).

Question sources

Computer Science MCQ practice collection · Question 466

Question 200 | AM0467

Given a consistent comparator on a priority field, which listed comparison sorting algorithms can sort records by that field? No particular extraction mechanism or equal-priority stability is required. Select all that apply.

A. Bubble sort

B. Quick sort

C. Merge sort

D. Heap sort

Hint

Compare the algorithmic steps and access policy with the exact resource or ordering requirement in the question.

Answer A, B, C, D: A. Bubble sort; B. Quick sort; C. Merge sort; D. Heap sort

Priority is a key field that any comparison sort can use through a consistent comparator. Bubble compares neighboring priorities, quicksort partitions by priority, mergesort combines priority-ordered runs, and heapsort repeatedly places a priority extreme. Since no heap-extraction mechanism or tie-stability condition is required, every listed algorithm can produce the requested priority ordering. Priorities alone do not identify one unique sorting method.

Why each choice is right or wrong

A - Correct: Bubble can compare adjacent priority fields.

B - Correct: Quicksort can partition with the priority comparator.

C - Correct: Mergesort can merge runs by the priority field.

D - Correct: Heapsort can extract priority extremes.

Primary note proof

Study-note section DSA38

> It is not uniquely the method for sorting by priorities: any comparison sort can compare a priority field. Heap extraction is its distinctive mechanism.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

> iii. Heap Sort
> Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
> each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
> Algorithm:
> 1. Construct a binary tree
> • The root node corresponds to Data[0].
> • If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
> 2. Construct the heap tree from initial binary tree using "adjust" process.
> Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Data Structure and Algorithms module | PDF page 8 | lines 40-45

> Priority Queue:- is a queue where each data has an associated key provided at insertion time.
> Application of Queue
> Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:

• Print server- maintains a queue of print jobs
• Disk Driver- maintains a queue of disk input/output requests
• Task scheduler in multiprocessing system- maintains priority queues of processes

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Data Structure and Algorithms module | PDF page 4 | lines 2-8

C. Bubble Sort
The simplest to implement but slowest algorithm on very large inputs. The algorithm loops through
the array from i=0 to n and swap adjacent elements if they are out of order bubbling or pushing out the smallest or largest element to its proper position.
• How many comparisons? => O(n2
) and How many swaps? => O(n2
)

Question sources

Computer Science MCQ practice collection · Question 467

Question 201 | AM0468

An LRU cache already has hash lookup that returns a reference to the matching entry. Which listed representation should maintain recency order so a known entry can be removed and moved to the recent end in constant time, with eviction at the least-recent end?

A. Array

B. Doubly linked list with head and tail

C. Hash table

D. Queue

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer B: B. Doubly linked list with head and tail

The hash index handles locating an entry, while the recency component must unlink a known entry and reconnect it at the recent end. A doubly linked list supplies both neighboring links for that constant-time removal and maintains endpoints for movement and eviction. An array would generally shift entries, a bare hash table does not supply recency order, and an ordinary FIFO queue cannot remove an arbitrary known interior entry.

Why each choice is right or wrong

A - Incorrect: Maintaining array recency order generally requires shifts.

B - Correct: A doubly linked recency list supports unlinking a known node and endpoint updates.

C - Incorrect: The existing hash index handles lookup but does not itself maintain LRU order.

D - Incorrect: An ordinary queue lacks arbitrary known-interior-node removal.

Primary note proof

Study-note section DSA27

> A dictionary or compiler symbol table can use hashing for expected constant lookup, or a balanced tree for logarithmic lookup with ordering. A bounded cache also needs eviction rules: FIFO uses a queue; LRU can combine hash lookup with a doubly linked recency list. Move a hit to the recent end and evict the least-recent end when full. No storage structure is universally best without operation and policy requirements.

Study-note section DSA10

> Doubly linked nodes add prev; head and tail enable constant-time operations at both ends. Singly linked tail deletion still needs its predecessor. A circular tail links back to head; stop traversal on returning to the start, handling empty separately.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 4 | lines 16-25

1. Single Linked lists
It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
Basic Operations on Single Linked List
I. Creating Linked Lists
Structure definition of a single linked list need to contain both data items declaration and next node
pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node added
to the chain.

Data Structure and Algorithms module | PDF page 5 | lines 13-24

Step 3: Then add the node at the end of the list. For this we need to check whether the list is empty or not. If list is empty the new node will be the first (Start) node, otherwise navigate forward starting form the first node till last node found and add it next to this node which make the new node the last node.
if (start == NULL) // If list is empty, then new node become start node
start = temp;
else {
temp2 = start; // list not empty! Thus temp2 starts from first node
while (temp2->next!= NULL) { // till the current last node found navigates forward
temp2 = temp2->next; // Move to next link in chain
}
temp2->next = temp; // new node (temp) appended at the end of the list

Data Structure and Algorithms module | PDF page 5 | lines 35-41

IV. Deleting a node from the list
When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the delete instruction:
delete temp; // Release the memory pointed to by temp
However, we can't just delete the nodes willy-nilly as it would break the chain. We need
to reassign the pointers and then delete the node at the last moment. We can delete nodes
at the start, middle or end of the list. Here let’s see the steps to delete at the end.

Data Structure and Algorithms module | PDF page 7 | lines 10-16

“A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT”. Queues are sometimes referred to as First in First out (FIFO) lists.
The two basic operations are:
Enqueue - Inserts an item / element at the rear end of the queue. An overflow error occurs
if the queue is full.
Dequeue - Removes an item / element from the front end of the queue, and returns it to
the user. An underflow error occurs if the queue is empty.

Question sources

Computer Science MCQ practice collection · Question 468

Question 202 | AM0469

Which algorithm is used to find the maximum subarray sum in an array?

A. Binary search

B. Quick sort

C. Merge sort

D. Kadane's algorithm

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer D: Kadane’s algorithm

DSA44 directly identifies Kadane as the contiguous maximum-sum-subarray algorithm. It updates the best sum ending at the current item and the best sum seen overall, using linear time and constant working storage. Binary search does not optimize contiguous sums, and quicksort or merge sort reorder values rather than preserve the original adjacency needed by this problem.

Why each choice is right or wrong

A - Incorrect: Binary search locates targets or ordered boundaries; it does not compute a contiguous maximum-sum subarray.

B - Incorrect: Quicksort reorders values and does not preserve the original adjacency required for maximum subarray sum.

C - Incorrect: Merge sort orders values; the named sorting algorithm is not the maximum-subarray recurrence.

D - Correct: Kadane maintains the best sum ending here and the best anywhere, directly solving the stated problem.

Primary note proof

Study-note section DSA44

> Kadane’s algorithm finds a contiguous maximum-sum subarray. For a required nonempty answer, initialize ending=best=A[0]; update ending=max(x,ending+x), then best=max(best,ending). It is linear time with constant working storage. Initializing best=0 instead silently allows an empty answer and fails the nonempty all-negative case.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 3 | lines 16-22

Linear/Sequential searching
• Works with unsorted List
• Loop through the list starting at the first element
• Time is proportional to the size of input (n) and we call this time complexity O(n).

Data Structure and Algorithms module | PDF page 11 | lines 14-22

ii. Quick sort: is the fastest known algorithm. It uses divide and conquer strategy and in the worst case its complexity is O (n2). But its expected complexity is O(nlogn).
Algorithm:
1. Choose a pivot value (mostly the first element is taken as the pivot value)
2. Position the pivot element and partition the list so that:
• the left part has items less than or equal to the pivot value
• the right part has items greater than or equal to the pivot value
3. Recursively sort the left part
4. Recursively sort the right part

Data Structure and Algorithms module | PDF page 11 | lines 38-44

iv. Merge sort
Merge sort uses divide and conquer strategy and its time complexity is O(nlogn).
Algorithm:
1. Divide the array in to two halves.
2. Recursively sort the first n/2 items.
3. Recursively sort the last n/2 items.
4. Merge sorted items (using an auxiliary array).

Question sources

Computer Science MCQ practice collection · Question 469

Question 203 | AM0470

Which data structure is commonly used to implement a graph?

A. Array

B. Linked list

C. Hash table

D. Adjacency list

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer D: Adjacency list

DSA40 describes adjacency lists as storing each vertex's neighbors in Θ(V+E) space. This names a complete graph representation among the choices. Arrays, linked lists, or maps may store parts of a representation, but those container names alone do not specify how vertices and their edges are encoded. The answer identifies the graph-specific organization.

Why each choice is right or wrong

A - Incorrect: An array can store a matrix or adjacency containers, but its name alone does not specify a graph representation.

B - Incorrect: A linked list can hold neighbor records, but alone does not describe the vertex-to-neighbor organization.

C - Incorrect: Hash tables may index vertices or neighbors, but do not by themselves name the graph representation supplied here.

D - Correct: An adjacency list explicitly associates vertices with their neighbors and is a standard graph representation.

Primary note proof

Study-note section DSA40

> A matrix uses Theta(V^2) space and constant edge lookup. Adjacency lists use Theta(V+E) space and enumerate neighbours. Dense simple graphs approach quadratic edge counts. Examining every edge costs more on denser graphs; vertex-only operations need not inspect every edge.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 2 | lines 15-18

> a. Linear data structure: data items stores in sequence
> Example: Array, stack and Queue,
> b. Non-Linear data structures : in which there is order of data items
> Example: Trees and Graphs

Data Structure and Algorithms module | PDF page 4 | lines 16-25

> 1. Single Linked lists
> It is a self-referential structure (Structures can hold pointers to instances of themselves), that allow
> to create a chain of structs/Nodes with related data. Each node in the list contains data items and pointer to next node.
> Basic Operations on Single Linked List
> I. Creating Linked Lists
> Structure definition of a single linked list need to contain both data items declaration and next node
> pointer. In addition the first node pointer is essential to keep track of the list to perform various operations on the list. Thus, it is initialized to NULL first as the list is empty before any node

added
to the chain.

Question sources

Computer Science MCQ practice collection · Question 470

Question 204 | AM0471

Which algorithm is used to find the minimum spanning tree in a graph?

A. Depth-first search

B. Breadth-first search

C. Dijkstra's algorithm

D. Prim's algorithm

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer D: Prim’s algorithm

DSA43 defines an MST as a minimum-total-cost connection of all vertices in a connected undirected weighted graph. Prim repeatedly adds the cheapest edge crossing from its current tree to an outside vertex. Dijkstra minimizes source-path distances instead, while BFS and DFS are traversals without a general minimum-weight spanning-tree guarantee. Thus Prim is the listed MST algorithm.

Why each choice is right or wrong

A - Incorrect: DFS can produce a spanning tree but does not generally minimize total edge weight.

B - Incorrect: BFS produces minimum-edge source layers in unweighted graphs, not a general minimum-weight spanning tree.

C - Incorrect: Dijkstra minimizes source-to-vertex distances, a different objective from total spanning-tree connection cost.

D - Correct: Prim grows a spanning tree by repeatedly choosing a cheapest edge crossing to an outside vertex.

Primary note proof

Study-note section DSA43

An MST minimizes the total cost of connecting all vertices of a connected undirected weighted graph. Prim adds the cheapest edge crossing from its current tree to an outside vertex. Kruskal processes edges by increasing weight and rejects those joining vertices already connected. An MST need not give shortest source-to-vertex routes, and equal weights can permit several MSTs.

Study-note section DSA43

For nonnegative weights, standard Dijkstra initializes source distance zero and others infinity, repeatedly finalizes the unsettled vertex with smallest distance, and relaxes each outgoing edge: dist[v]=min(dist[v],dist[u]+w(u,v)). A min-priority queue supports the next choice. BFS is enough when every edge has equal cost; negative edges require another justified method, such as Bellman-Ford.

Question sources

Computer Science MCQ practice collection · Question 471

Question 205 | AM0472

A priority queue needs O(1) peek and O(log n) worst-case insertion and extraction for an interleaved workload. Compare ordinary sorted/unsorted array and linked-list implementations, a binary heap, and an unmodified FIFO queue; exclude representations with extra search indexes. Which meets all requirements?

A. Array

B. Linked list

C. Binary heap

D. Queue

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer C: C. Binary heap

A binary heap exposes the extreme key at its root, so peek is constant, and insertion or extraction restores order along a logarithmic-height path. Ordinary unordered arrays or lists need a linear search for an extreme, while ordered versions pay linear insertion in the general case. An unmodified FIFO queue follows arrival order rather than priority. The stated operation bounds make the heap the unique matching standard representation.

Why each choice is right or wrong

A - Incorrect: Ordinary ordered/unsorted arrays trade linear insertion against linear extremum selection.

B - Incorrect: Ordinary lists likewise have a linear cost in one required operation.

C - Correct: A binary heap provides constant peek and logarithmic insertion/extraction.

D - Incorrect: An unmodified queue serves FIFO rather than priority order.

Primary note proof

Study-note section DSA17

A priority queue returns the smallest or largest priority key under its convention. Urgent patients may precede earlier arrivals. Insertions and extractions can interleave. Binary heaps support constant peek and logarithmic insert/extract.

Study-note section DSA17

Arrays and lists can also implement priority queues: an unordered sequence gives cheap insertion but linear extremum removal; an ordered sequence reverses that tradeoff. Equal priorities have no inherent FIFO guarantee. If stable service is required, compare (priority, arrival_counter). Choose the priority direction and tie rule before tracing.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 11 | lines 23-37

iii. Heap Sort
Heap sort operates by first converting the list in to a heap tree. Heap tree is a binary tree in which
each node has a value greater than both its children (if any). It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent. The time complexity of heap sort is O(nlogn).
Algorithm:
1. Construct a binary tree
• The root node corresponds to Data[0].
• If we consider the index associated with a particular node to be i, then the left child of this node corresponds to the element with index 2*i+1 and the right child corresponds to the element with index 2*i+2. If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.
2. Construct the heap tree from initial binary tree using "adjust" process.
Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

Data Structure and Algorithms module | PDF page 8 | lines 40-45

Priority Queue:- is a queue where each data has an associated key provided at insertion time.
Application of Queue
Queue is goodfor fair (firstcome first served) ordering of actions. Some of queue applications:
• Print server- maintains a queue of print jobs
• Disk Driver- maintains a queue of disk input/output requests
• Task scheduler in multiprocessing system- maintains priority queues of processes

Question sources

Computer Science MCQ practice collection · Question 472

Question 206 | AM0473

Which algorithm is commonly used for pattern matching in strings?

A. Depth-first search

B. Breadth-first search

C. Dijkstra's algorithm

D. Knuth-Morris-Pratt algorithm

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer D: Knuth-Morris-Pratt

DSA44 explicitly defines KMP as finding a pattern inside text. It preprocesses longest proper prefix/suffix lengths and uses them to avoid restarting comparisons after mismatch, giving linear combined pattern-and-text work. DFS, BFS, and Dijkstra concern graph exploration or paths, rather than this specialized substring-matching mechanism. KMP uniquely matches the question among these choices.

Why each choice is right or wrong

A - Incorrect: DFS explores graph or tree branches; it is not the substring-pattern algorithm identified in DSA44.

B - Incorrect: BFS explores graph distance layers rather than applying prefix/suffix reuse for text matching.

C - Incorrect: Dijkstra computes nonnegative weighted shortest paths, not substring pattern occurrences.

D - Correct: KMP is the pattern-matching algorithm using preprocessed prefix/suffix lengths to avoid repeated text comparisons.

Primary note proof

Study-note section DSA44

> Knuth-Morris-Pratt (KMP) finds a pattern inside text. Preprocess the pattern’s longest proper prefix that is also a suffix. After mismatch, fall back using those prefix lengths rather than restarting comparisons from the beginning; the text pointer need not move backward. Pattern preprocessing plus scanning takes O(pattern length+text length).

Question sources

Computer Science MCQ practice collection · Question 473

Question 207 | CS0122

Assume a very large finite key universe U but only n stored keys, with n much smaller than |U|. A hash table uses capacity proportional to n. Compare allocated storage, rather than unconditional speed. Which statement is false?

A. For n much smaller than |U|, the specified hash table uses less allocated storage than direct addressing.

B. Direct Address table doesn't use hash function to map keys

C. Hash table allocates one separate memory slot for each key in the universe U

D. There is one-to-one correspondence between keys in universe U and memory slots in the Direct Address table

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer C: C. Hash table allocates one separate memory slot for each key in the universe U

Direct addressing reserves a distinct slot for every possible key in the universe, whether present or absent, so its storage is proportional to |U|. The specified hash table stores the actual entries plus capacity proportional to n. For this sparse-universe storage comparison, hashing uses less space. C incorrectly attributes the direct-address allocation rule to hashing, while direct key-to-slot mapping and separate universe slots accurately describe direct addressing.

Why each choice is right or wrong

A - Incorrect: Under the sparse-universe storage assumptions, the hash table uses fewer allocated slots.

B - Incorrect: Direct addressing maps directly to a key’s universe slot without compressive hashing.

C - Correct: One slot for every possible key describes direct addressing, not the specified hash table.

D - Incorrect: Each universe key has a separate direct-address slot.

Primary note proof

Study-note section DSA25

> Direct addressing reserves a position for every possible key in a finite universe U, costing Theta(|U|) slots. A hash table uses h(key) in 0..m-1 and stores only actual entries plus bucket capacity. With h(k)=k mod 6, key 36 has home address 0; collisions may move its final storage position.

Question sources

Computer Science Exit Exam practice collection · Question 122

Question 208 | CS0131

What is the time complexity order of push() operation of Stack data structure implemented using array?

A. O(n)

B. O(log n)

C. O(n^2)

D. O(1)

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer D: O(1)

The fixed-array push checks fullness, increments top and stores one item. A bounded number of unit-cost operations independent of n gives O(1). Evidence: assigned PDF pages 6.

Why each choice is right or wrong

A - Incorrect: Linear would require scanning or copying that the given push does not do. See assigned PDF 6.

B - Incorrect: Logarithmic would require a search/adjustment absent from push. See assigned PDF 6.

C - Incorrect: Quadratic has no nested work in this code. See assigned PDF 6.

D - Correct: Only a fullness test, index increment and one assignment are performed. See assigned PDF 6.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA11

> A count-based alternative uses size=0 and A[size++]=x. Do not combine its empty/full tests with the top-index convention. Fixed-array push/pop are constant time. Linked stacks achieve constant time by inserting and deleting at the head; using a singly linked tail for pop would require a predecessor traversal.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 6 | lines 24-42

I. Array implementation (static memory).
Push(n): Adding item at top
top is the current top of stack and n is its
maximum size
Begin
if (top == n) then stack is full;
else
top := top+1;
stack[top] := item;
end:
Pop(): removing item from stack top
remove top element from the stack and put it in
the item
Begin:
if (top == 0) then stack is empty;
else
item := stack[top];
top := top-1;
end;

Question sources

Computer Science Exit Exam practice collection · Question 131

Question 209 | CS0132

A zero-based six-slot hash table uses h(key)=key%6. What home index does it compute for key 36, before any collision resolution?

A. At index 35

B. At index 1

C. At index 6

D. At index 0

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer D: D. At index 0

The home address is obtained by applying the stated remainder function. Since 36 is exactly divisible by 6, the remainder is zero and the home index is 0. The question stops before collision resolution, so occupied slots cannot change this computed address. Index 1 could be a displaced location under some probing state, but it is not the function's output; indices 6 and 35 are outside the table.

Why each choice is right or wrong

A - Incorrect: 35 is outside the valid zero-based range 0-5.

B - Incorrect: 1 is not the remainder of 36 divided by 6.

C - Incorrect: 6 is outside the valid range and is not a possible remainder modulo 6.

D - Correct: 36%6 equals 0.

Primary note proof

Study-note section DSA25

> Direct addressing reserves a position for every possible key in a finite universe U, costing Theta(|U|) slots. A hash table uses h(key) in 0..m-1 and stores only actual entries plus bucket capacity. With h(k)=k mod 6, key 36 has home address 0; collisions may move its final storage position.

Question sources

Computer Science Exit Exam practice collection · Question 132

Question 210 | CS0133

Which method can resolve collision in hashed files by proceeding from the occupied position specified by the hash address and checking subsequent positions until an empty position is found?

A. Open addressing

B. Multiple hashing

C. Chaining

D. Cluster indexing

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer A: Open addressing, specifically linear probing.

DSA26 defines open addressing as keeping entries within the array and probing until an available slot is found. Checking successive positions is its linear-probing form. Multiple hashing changes probe steps, chaining stores collisions in bucket lists, and cluster indexing is not this insertion procedure. Therefore Open addressing is the matching general method among the options.

Why each choice is right or wrong

A - Correct: Open addressing searches alternative table positions; checking successive positions is its linear-probing form.

B - Incorrect: Multiple or double hashing derives another probe step rather than the stated simple successive-slot scan.

C - Incorrect: Chaining puts colliding entries in a bucket list instead of searching successive array slots.

D - Incorrect: Cluster indexing does not name the collision insertion strategy described by the successive-position probing rule.

Primary note proof

Study-note section DSA26

> Separate chaining stores colliding entries in a bucket list. Load alpha=n/m is average chain length; expected operations are O(1+alpha) under suitable distribution assumptions, with O(n) worst cases. Open addressing keeps entries inside the array and probes until a key or an available slot is found.

Study-note section DSA26

> Linear probing tries (h(k)+j)%m. For size 7 and keys 10,17,24, all home addresses are 3 and occupied positions become 3,4,5. Quadratic probing changes the step; double hashing uses h1(k)+j h2(k), modulo m, with a step relatively prime to m to cover all slots. Neither method guarantees no collisions. Deletion needs tombstones or reinsertion so searches do not stop prematurely.

Question sources

Computer Science Exit Exam practice collection · Question 133

Question 211 | CS0134

Which one of the following is true about tree data structure?

A. Tree is a static data structure

B. Tree is a linear data structure

C. Tree forms a cycle - in other words tree is a cyclic data structure

D. There is only one unique path when we traverse from one node to another node in a tree

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer D: There is only one unique path when we traverse from one node to another node in a tree

A rooted tree gives every nonroot node exactly one parent and a unique root path. Between any two nodes, ascend to their common ancestor and descend; a second distinct path would create a cycle and violate the parent/path hierarchy. Evidence: assigned PDF pages 9, 2.

Why each choice is right or wrong

A - Incorrect: The pointer-based tree representation does not require static size. See assigned PDF 9, 2, 9.

B - Incorrect: The source explicitly classifies trees as nonlinear. See assigned PDF 9, 2, 9.

C - Incorrect: A cycle contradicts the unique-path hierarchy. See assigned PDF 9, 2, 9.

D - Correct: The rooted hierarchy gives one unique connecting path for every node pair. See assigned PDF 9, 2, 9.

Primary note proof

Study-note section DSA18

> A rooted tree has one parentless root and one parent per other node. It is connected and acyclic, has n-1 edges, and one path between any pair. Depth counts root-to-node edges; height is maximum depth. A single-node tree therefore has height zero. Leaves have no children.

Module passages - related reading

These source passages provide context; the primary note proof is identified separately above.

Data Structure and Algorithms module | PDF page 9 | lines 3-13

> A tree is a set of nodes and edges that connect pairs of nodes that connect pairs of nodes. It is an abstract model of a hierarchical structure. Rooted tree has the following structure:
> • One node distinguished as root.

• Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
• There is a unique path from the root to the each node.
• The number of edges in a path is the length of the path.
Tree Terminologies:
• Root: a node without a parent.
• Internal node: a node with at least one child.
• External (leaf) node: a node without a child.

Data Structure and Algorithms module | PDF page 2 | lines 15-18

a. Linear data structure: data items stores in sequence
Example: Array, stack and Queue,
b. Non-Linear data structures : in which there is order of data items
Example: Trees and Graphs

Data Structure and Algorithms module | PDF page 9 | lines 19-25

• Binary tree: a tree in which each node has at most two children called left child and right child.
• Binary search tree (ordered binary tree): a binary tree that may be empty, but if it is not empty it
satisfies the following.
o Every node has a key and no two elements have the same key.
o The keys in the right subtree are larger than the keys in the root.
o The keys in the left subtree are smaller than the keys in the root.
o The left and the right subtrees are also binary search trees.

Question sources

Computer Science Exit Exam practice collection · Question 134

Question 212 | CS0135

For an unordered heap record file without an ordered index or a maintained sorted copy, which statement is not correct? Ordinary insertion uses available storage rather than maintaining field order.

A. Sorting is required to read records in particular field order

B. Linear search can be used to search a necessary record

C. Record insertion is efficient

D. New records are inserted at the beginning of the file

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer D: D. New records are inserted at the beginning of the file

A heap file does not maintain record order by a chosen field. With no ordered index or sorted copy, ordered output therefore requires sorting; an ordinary scan can locate a record, and insertion can place a record in available storage. Nothing requires new records to be inserted at the beginning of the file. D is the incorrect claim, while the explicit no-index scope makes A's ordering requirement appropriate.

Why each choice is right or wrong

A - Incorrect: Under the specified no-index/no-sorted-copy scope, field-ordered output requires ordering the records.

B - Incorrect: A linear scan can search unordered records.

C - Incorrect: Using available storage avoids maintaining a sorted record sequence during insertion.

D - Correct: Heap-file insertion is not required to place every new record at the beginning.

Primary note proof

Study-note section DSA45

> Specify whether a question asks which structure can implement something or which is best under stated costs. ADTs do not force one representation, and serial-access memory is not inherently LIFO. A heap file is an unordered database record file, unrelated to a binary heap: insertion uses available space, scanning can find records, and ordered output requires sorting or an index. It need not insert at the beginning.

Question sources

Computer Science Exit Exam practice collection · Question 135

Question 213 | NS0298

Which listed structures are linear in the course classification? Select all that apply.

A. Tree

B. Array

C. Queue

D. Linked list

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer B, C, D: B. Array; C. Queue; D. Linked list

A linear structure arranges items along a sequence rather than through hierarchical branching. The note explicitly includes arrays, queues and linked lists in that category, giving three valid selections. A tree instead models a hierarchy and is classified as nonlinear. The select-all instruction allows every listed member of the linear category to be recognized rather than forcing an arbitrary single choice.

Why each choice is right or wrong

A - Incorrect: Trees model hierarchy rather than a linear sequence.

B - Correct: Arrays are classified as linear.

C - Correct: Queues are classified as linear.

D - Correct: Linked lists are classified as linear.

Primary note proof

Study-note section DSA01

> An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Question sources

Computer Science Exit Exam · 2018 · Question 15

Question 214 | NS0319

Which abbreviations name the stack service policy? Select all that apply.

A. LIFO

B. FIFO

C. FILO

D. FCFS

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer A, C: A. LIFO; C. FILO

The most recently pushed item is removed first by a stack. This is last in, first out, abbreviated LIFO. Equivalently, the first item inserted remains until later items have been removed, making it first in, last out or FILO. FIFO and FCFS instead preserve arrival order in ordinary queue service. Both A and C therefore name the same stack discipline.

Why each choice is right or wrong

A - Correct: LIFO means last in, first out.

B - Incorrect: FIFO means first in, first out, an ordinary queue policy.

C - Correct: FILO is the equivalent first in, last out view of a stack.

D - Incorrect: FCFS describes arrival-order service rather than stack service.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA14

> An ordinary queue enqueues at rear, dequeues at front, and peeks at front. This is FIFO or first-come, first-served. A linked implementation with head and tail supports both operations in constant time: append at tail, remove at head. The first insertion sets both pointers; removing the last item clears both. With head alone, insertion must traverse to the tail.

Question sources

Computer Science Exit Exam · 2018 · Question 44

Question 215 | NS0337

For ordinary binary search on a sorted random-access array of n elements, what is the tightest listed worst-case time bound? Count each indexed probe as constant work.

A. $O(n (\log n)^2)$

B. $O(n^2)$

C. O(n)

D. O(log n)

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer D: D. O(log n)

Binary search repeatedly removes roughly half of the remaining candidate interval. The note bounds its worst probes by $\text{floor}(\log_2 n)+1$ for nonempty input, giving logarithmic worst-case time under constant-cost indexing. The linear, quadratic and n-times-squared-log options all grow faster and are looser upper bounds. Explicitly asking for the tightest bound selects the logarithmic alternative rather than treating every valid Big-O upper bound as equally responsive.

Why each choice is right or wrong

A - Incorrect: $n(\log n)^2$ is a much looser upper bound.

B - Incorrect: Quadratic is a looser upper bound.

C - Incorrect: Linear is a looser upper bound.

D - Correct: Logarithmic matches the interval-halving probe growth.

Primary note proof

Study-note section DSA29

> Worst probes are $\text{floor}(\log_2 n)+1$ for $n>0$: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Study-note section DSA06

> For eventually nonnegative functions, $f(n)=O(g(n))$ means $f(n) \le c\, g(n)$ for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Question sources

Computer Science Exit Exam · 2018 · Question 68

Question 216 | NS0351

Which sorting algorithm compares each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swaps them; if at least one swaps has been done, repeat step 1.

A. Insertion sort

B. Bubble sort

C. Selection sort

D. Shell sort

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer B: Bubble sort with early stopping

DSA32 describes adjacent comparisons, swapping strict inversions, and repeating passes until a pass performs no swaps. That matches every part of the stem, including its stopping condition. Insertion grows a sorted prefix, selection chooses a suffix minimum, and Shell uses separated gap subsequences, so none has this exact adjacent full-pass mechanism.

Why each choice is right or wrong

A - Incorrect: Insertion places each next value into a sorted prefix rather than repeating full adjacent passes.

B - Correct: Adjacent inversion swaps and repetition until no swaps identify optimized bubble sort.

C - Incorrect: Selection finds a suffix minimum and exchanges it with the prefix boundary.

D - Incorrect: Shell sorts gap-separated subsequences before its final gap-one insertion pass.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right. The largest remaining value moves to the final unsorted position after one left-to-right pass. Shrink the next pass by one position. Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Question sources

Computer Science Exit Exam · 2018 · Question 87

Question 217 | NS0354

Among the following one is NOT the application of queue in the real world.

A. It is used in operating systems for handling interrupts.

B. It is used to maintain the play list in media players in order

C. It is used in Depth First search

D. It is widely used as waiting lists for a single shared resource

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer C: Depth-first search uses stack discipline rather than the ordinary queue.

DSA42 finishes a branch using a stack or recursion and explicitly says a queue changes that discipline. DSA14 gives FIFO waiting, play sequences, and asynchronous buffering as queue uses. Interrupt work can also be buffered pending service. Thus the standard DFS claim is the non-queue application among the offered ordinary examples.

Why each choice is right or wrong

A - Incorrect: Pending interrupts can be buffered for later service; queue use need not imply one universal interrupt priority policy.

B - Incorrect: A FIFO play queue can preserve the order of media items.

C - Correct: Standard DFS uses latest-pending-work-first stack or recursion rather than FIFO queue service.

D - Incorrect: Waiting for one shared resource is a direct FIFO queue application.

Primary note proof

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Study-note section DSA14

> FIFO suits equal-priority printer jobs, shared-resource waiting, asynchronous buffers and level-order exploration. A play sequence can also use a queue. Distinguish airline seat records from waiting requests; an application name alone does not force one representation.

Study-note section DSA-S02

> An interrupt service routine may need to record work quickly and defer its completion. Operating systems can queue that deferred work for later processing; Windows deferred procedure calls are one concrete example. This is a queue application, but it does not imply that every interrupt system uses a single FIFO service policy.

Question sources

Computer Science Exit Exam · 2018 · Question 91

Question 218 | NS0519

Which data structure uses LIFO?

A. Queue

B. Array

C. Stack

D. Linked List

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer C: Stack

DSA11 explicitly assigns LIFO to a stack. Arrays and linked lists can represent that stack, but neither storage structure inherently fixes its removal order. A queue instead supplies FIFO. The question asks for the service-policy ADT, so Stack is the specific answer rather than one of the possible underlying representations.

Why each choice is right or wrong

A - Incorrect: A queue follows FIFO, not the latest-in-first-out order requested.

B - Incorrect: An array is a representation and does not inherently impose LIFO service.

C - Correct: A stack is explicitly defined by LIFO, equivalently FILO.

D - Incorrect: A linked list can implement several policies and does not inherently require LIFO.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA01

> An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Question sources

Data Structures & Algorithms review questions · Question 1

Question 219 | NS0520

What distinguishes a dynamic data structure from a static one?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Dynamic structures can change size at runtime; static structures have fixed capacity.

DSA02 distinguishes fixed arrays from resizable arrays, and DSA08 allocates linked nodes as needed. Runtime growth or shrinkage is the key distinction; fixed capacity need not mean compile-time allocation.

Primary note proof

Study-note section DSA02

> A fixed array has a chosen capacity; the choice need not occur at compile time. A resizable array allocates a larger block and copies when needed. Inserting in the middle shifts later elements, so it costs linear time. Appending into spare capacity is constant; a resizing append can be linear, though repeated geometric growth gives amortized constant append cost.

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Question sources

Data Structures & Algorithms review questions · Question 2

Question 220 | NS0521

Which classification includes both Trees and Graphs?

A. Linear

B. Homogeneous

C. Non-linear

D. Static

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer C: Non-linear

DSA01 treats arrays, lists, stacks, and queues as linear sequences, trees as hierarchies, and graphs as general connections. Trees and graphs therefore belong to the nonlinear relationship category. Homogeneity describes element types and static describes storage capacity; neither classifies both by their branching relationship structure. C is the appropriate shared category.

Why each choice is right or wrong

A - Incorrect: Linear describes sequence-like structures rather than branching trees and general graphs.

B - Incorrect: Homogeneous concerns uniform element types and does not identify their relationship structure.

C - Correct: Trees and graphs are nonlinear relationship structures, unlike the listed linear sequences.

D - Incorrect: Static concerns storage or capacity decisions, not whether connections branch.

Primary note proof

Study-note section DSA01

> An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Question sources

Data Structures & Algorithms review questions · Question 3

Question 221 | NS0522

In an ADT, what two components are always defined?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: An ADT defines its values/data and permitted operations.

DSA01 explicitly defines an abstract data type through values and operations, leaving representation to the implementation. The same LIFO stack can therefore use arrays or linked nodes without changing its contract.

Primary note proof

Study-note section DSA01

> An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Question sources

Data Structures & Algorithms review questions · Question 4

Question 222 | NS0523

A structure in C that groups int, char, and float is an example of:

A. Homogeneous

B. Non-homogeneous

C. Linear

D. Static

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer B: Non-homogeneous, or heterogeneous

DSA01 directly identifies a record grouping integer, character, and floating fields as heterogeneous. The different field types distinguish it from an ordinary same-element-type array. Linear describes relationships, and static describes storage behavior, neither of which follows merely from the field types. Thus Non-homogeneous is the requested classification.

Why each choice is right or wrong

A - Incorrect: Homogeneous would mean a uniform element type, contrary to the mixed int, char, and float fields.

B - Correct: The record combines different field types and is therefore heterogeneous or non-homogeneous.

C - Incorrect: Linear concerns relationships or order, not the mixed-type classification asked here.

D - Incorrect: Static concerns allocation or lifetime and is not established merely by these field types.

Primary note proof

Study-note section DSA01

> Arrays, linked lists, stacks and queues are linear. Trees model hierarchies; graphs model general connections. A record grouping integer, character and floating fields is heterogeneous; an ordinary array has one element type.

Question sources

Data Structures & Algorithms review questions · Question 5

Question 223 | NS0524

What does abstraction in algorithms allow programmers to do?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Abstraction preserves relevant behavior and hides irrelevant implementation detail.

DSA01 illustrates abstraction using patient urgency and arrival order while ignoring hair colour. Programmers retain the data and operations needed for the task and separate that contract from representation details. This supports the substantive answer without additional facts.

Primary note proof

Study-note section DSA01

> An abstract data type (ADT) defines values and operations; representation supplies implementation. Arrays or linked nodes can implement the same LIFO stack. Abstraction keeps relevant details: patient urgency and arrival order matter for service; hair colour does not.

Question sources

Data Structures & Algorithms review questions · Question 6

Question 224 | NS0525

Which operation involves combining elements from two sorted sets into one sorted set?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Merging

DSA03 calls combining sequences merging and works through sorted inputs by repeatedly emitting the smaller current front. Advancing only the selected sequence preserves sorted output until the remaining suffix is appended. For mathematical sets, duplicate handling additionally follows the set contract.

Primary note proof

Study-note section DSA03

> Traversal processes every item; searching locates a target; insertion and deletion change membership; updating changes a stored value. Sorting reorders items, while merging combines sequences. To merge sorted [1,4] and [2,3], repeatedly emit the smaller front, producing [1,2,3,4]. Destroying a structure releases owned storage.

Study-note section DSA27

> A set forbids duplicate elements.

Question sources

Data Structures & Algorithms review questions · Question 7

Question 225 | NS0526

What is the term for the operation that releases memory when a data structure is no longer needed?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Destruction or deallocation of owned storage.

DSA03 says destroying a structure releases owned storage. In a linked chain, DSA09 visits and releases each owned node; C++ allocation forms must be matched to their deletion forms, and freed nodes must not be dereferenced.

Primary note proof

Study-note section DSA03

> A C struct groups named fields, including arrays and pointers. A singly linked node needs payload plus a same-type link. C fields cannot have initializers inside the type declaration; modern C++ permits default member initializers. In C++, match new with delete and new[] with delete[]. Repair links before freeing a node; never dereference it afterwards.

Study-note section DSA09

> Deleting the tail of a singly linked list requires its predecessor, even when tail itself is known. Handle empty and single-node lists separately; after removing the only node, set both head and tail to null. To remove a known successor, bypass it before deallocation. Clearing an owned n-node chain visits and releases every node.

Question sources

Data Structures & Algorithms review questions · Question 8

Question 226 | NS0527

Given T(n) = 6n + 4, what is the Big-O complexity?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: O(n), tightly Θ(n).

For $n \geq 1$, $6n+4 \leq 10n$, while $6n+4 \geq 6n$. These constant upper and lower multiples establish linear growth by DSA06. O(n) is the natural tightest-order response, though larger upper bounds also exist.

Primary note proof

Study-note section DSA06

> For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Question sources

Data Structures & Algorithms review questions · Question 9

Question 227 | NS0528

Show why $f(n) = 3n^2 + 4n + 1$ is $O(n^2)$. What are the values of c and k?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: One valid witness pair is c=8, k=1.

For every $n \geq 1$, $4n \leq 4n^2$ and $1 \leq n^2$, so $3n^2+4n+1 \leq 8n^2$. DSA06 supplies the same inequality, proving the required eventual upper bound with c=8 and k=1. Other larger constants may also work.

Primary note proof

Study-note section DSA06

> For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Question sources

Data Structures & Algorithms review questions · Question 10

Question 228 | NS0529

Which property of an algorithm ensures it terminates after a finite number of steps?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Finiteness

DSA04 defines finiteness as terminating after finitely many steps for every valid input. It is distinct from definiteness, executable instructions, or correctness: a terminating procedure can still return a wrong answer. The required concept is directly covered.

Primary note proof

Study-note section DSA04

> Finiteness means termination after finitely many steps for every valid input. Definiteness means each instruction has one interpretation; effectiveness means each step is executable. Correctness means meeting the stated result for every valid input. A finite program can still return the wrong answer.

Question sources

Data Structures & Algorithms review questions · Question 11

Question 229 | NS0530

An algorithm has exactly two independent nested loops. Each executes n iterations, and the innermost body performs constant work. Loop-control operations are constant time, with no early exits or hidden nonconstant operations. What is the tight time complexity as n grows?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Theta(n^2) time.

For each of the n outer iterations, the inner loop executes n times, so the constant-cost body runs $n \times n = n^2$ times. Outer-loop overhead is linear and inner-loop control is also bounded by a constant per body execution. Those contributions do not change the dominant quadratic growth. Under the stated independence, fixed loop count and body-cost assumptions, the tight time complexity is Theta(n^2).

Primary note proof

Study-note section DSA07

> A loop i=1; i<n; i*=2 takes about $\log_2 n$ iterations. Two independent n-iteration loops nested around constant work take Theta(n^2). An n-level ordinary recursion with constant live storage per level needs Theta(n) stack space.

Study-note section DSA05

> Time counts chosen operations; space counts memory, including live recursive frames. Distributed algorithms may also measure communication volume. A single assignment is one unit in the elementary model, not literally one second. Sequential blocks add costs; independent nested loops multiply their iteration counts. Conditional worst-case cost includes the test and the more expensive branch.

Question sources

Data Structures & Algorithms review questions · Question 12

Question 230 | NS0531

Big-Theta (Θ) notation represents:

A. Upper bound only

B. Lower bound only

C. Both upper and lower bound

D. Average case only

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer C: Both upper and lower asymptotic bounds.

DSA06 defines Big-O as an eventual upper bound, Omega as the reverse bound, and Theta as requiring both. Thus Theta expresses a tight growth class up to positive constants. It is independent of choosing best, average, or worst input cases; Average case only confuses the bound relation with an input-distribution choice.

Why each choice is right or wrong

A - Incorrect: An upper bound alone is Big-O and does not supply Theta's lower requirement.

B - Incorrect: A lower bound alone is Omega and does not supply Theta's upper requirement.

C - Correct: Theta requires simultaneous eventual upper and lower constant-factor bounds.

D - Incorrect: Theta can describe any case or resource function, not only an average case.

Primary note proof

Study-note section DSA06

> For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Question sources

Data Structures & Algorithms review questions · Question 13

Question 231 | NS0532

Which grows faster asymptotically: O(n log n) or O(n)?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Representative n log n grows faster than n; Big-O classes themselves overlap.

DSA06 orders representative rates with n before n log n. Their ratio is log n, which grows without bound, so the representative n log n rate grows faster. However an O(n) function is also O(n log n), so membership alone does not compare actual runtimes.

Primary note proof

Study-note section DSA06

> For eventually nonnegative functions, f(n)=O(g(n)) means f(n)≤c g(n) for some positive c beyond a threshold k. Omega reverses the inequality; Theta requires both. Little-o means f/g tends to zero; little-omega means it grows without bound. These relations can describe best-case, average-case, worst-case or space functions.

Study-note section DSA06

> Increasing representative rates: 1, log n, n, n log n, n^2, n^3, 2^n, 3^n, n!, n^n. Logarithm bases differ only by a constant factor.

Question sources

Data Structures & Algorithms review questions · Question 14

Question 232 | NS0533

What is the running time contribution of a single assignment statement according to analysis rules?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: One constant time unit in the elementary-operation model.

DSA05 explicitly treats a single assignment as one unit in an elementary model, not one physical second. The convention abstracts hardware and requires that evaluating the assigned expression itself be constant cost; complicated expressions need their own accounting.

Primary note proof

Study-note section DSA05

> Time counts chosen operations; space counts memory, including live recursive frames. Distributed algorithms may also measure communication volume. A single assignment is one unit in the elementary model, not literally one second. Sequential blocks add costs; independent nested loops multiply their iteration counts. Conditional worst-case cost includes the test and the more expensive branch.

Question sources

Data Structures & Algorithms review questions · Question 15

Question 233 | NS0534

State the Transitivity theorem of Big-O. Why is it useful?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: If f∈O(g) and g∈O(h), then f∈O(h); this composes upper bounds.

DSA06 composes the eventual inequalities f≤c g and g≤d h to obtain f≤cd h beyond both thresholds. Choose the larger threshold and multiply the positive witness constants. This lets a complicated bound pass through a simpler intermediate function.

Primary note proof

Study-note section DSA06

> For n≥1, $3n^2+4n+1 \le 8n^2$: c=8,k=1 witness $O(n^2)$. Constants and lower-order terms disappear from tight polynomial growth. Transitivity multiplies witnesses: f≤c g≤cd h beyond both thresholds.

Question sources

Data Structures & Algorithms review questions · Question 16

Question 234 | NS0535

Rank from slowest to fastest growing: $O(n^2)$, $O(1)$, $O(2^n)$, $O(\log n)$, $O(n)$

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Increasing representative growth: 1, log n, n, n², 2^n.

DSA06 gives the standard growth ordering, with constants below logarithms, then linear and quadratic functions, then exponential. The ranking compares representative rates, not disjoint Big-O classes, since loose upper bounds overlap.

Primary note proof

Study-note section DSA06

> Increasing representative rates: 1, log n, n, n log n, n^2, n^3, 2^n, 3^n, n!, n^n. Logarithm bases differ only by a constant factor.

Question sources

Data Structures & Algorithms review questions · Question 17

Question 235 | NS0536

What are the three main resources considered in algorithm analysis?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Time, memory space, and-in distributed settings-communication volume.

DSA05 measures operation counts and live memory, and adds communication volume for distributed algorithms. Thus the third resource is context-dependent rather than a universal resource needed by every local algorithm. These are the three supported categories in the note.

Primary note proof

Study-note section DSA05

> Time counts chosen operations; space counts memory, including live recursive frames. Distributed algorithms may also measure communication volume. A single assignment is one unit in the elementary model, not literally one second. Sequential blocks add costs; independent nested loops multiply their iteration counts. Conditional worst-case cost includes the test and the more expensive branch.

Question sources

Data Structures & Algorithms review questions · Question 18

Question 236 | NS0537

What is the worst-case time complexity of Linear Search and when does it occur?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Θ(n) worst time, when absent or at the last position.

DSA28 scans until equality or exhaustion. An absent target or one in the final position requires all n comparisons, giving linear worst time; a first-position match instead takes constant time. The operation and cases are directly supported.

Primary note proof

Study-note section DSA28

> Compare the target with successive elements until equal or exhausted. Unsorted data is allowed; sorted data is allowed too. A first-element match costs Theta(1). An absent target or last-position match needs n comparisons, giving Theta(n) worst time. Uniformly likely successful positions average (n+1)/2 comparisons, still linear.

Question sources

Data Structures & Algorithms review questions · Question 19

Question 237 | NS0538

Binary Search requires the list to be:

A. Unsorted

B. Sorted

C. Circular

D. Linked

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer B: Sorted

DSA29 eliminates half an ordered interval according to the midpoint comparison. That elimination is valid only when comparable values follow a consistent order; descending input works with reversed comparisons. Circularity and links are representation properties, not substitutes for sortedness. An unsorted sequence gives no safe smaller-half decision. Therefore Sorted is the required prerequisite.

Why each choice is right or wrong

A - Incorrect: Without order, a midpoint comparison cannot safely eliminate either half.

B - Correct: Consistent sorted order supports the midpoint half-elimination invariant.

C - Incorrect: Circular storage does not establish the value ordering needed by binary search.

D - Incorrect: Linked storage does not imply sortedness and also changes midpoint-access costs.

Primary note proof

Study-note section DSA29

> On an ascending random-access sequence, set lo=0, hi=n-1. Compute mid=lo+(hi-lo)//2. Equality finishes; if target<A[mid], set hi=mid-1; otherwise set lo=mid+1. Stop unsuccessfully when lo>hi. Reverse comparisons for descending order. Comparable keys and valid finite bounds matter; fixed allocation before population is not an extra algorithmic requirement.

Question sources

Data Structures & Algorithms review questions · Question 20

Question 238 | NS0539

How many steps does Binary Search take for a list of 1,000,000 elements (approximately)?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: About 20 worst-case midpoint probes.

DSA29 gives floor($\log_2 n$)+1 probes for n>0. Since 2^19=524288 and 2^20=1048576, one million lies between them, so the worst probe count is twenty under the ordinary interval algorithm. A root/midpoint hit can finish earlier.

Primary note proof

Study-note section DSA29

> Worst probes are floor($\log_2 n$)+1 for n>0: seven for 100 keys, about twenty for one million. Iteration uses constant space; recursion uses logarithmic stack depth. Linked midpoint traversal loses logarithmic total time. Boundary variants retain a candidate rather than stopping at equality.

Question sources

Data Structures & Algorithms review questions · Question 21

Question 239 | NS0540

What is the time complexity of Bubble Sort and why?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Worst and usual average $\Theta(n^2)$; early-stop best $\Theta(n)$.

DSA32 sums repeated adjacent passes to at most n(n−1)/2 comparisons, yielding quadratic worst and ordinary average work. A no-swap early-stop pass makes sorted input linear, while a fixed pass schedule remains quadratic. The answer needs these implementation and case distinctions.

Primary note proof

Study-note section DSA32

> For ascending order, compare each adjacent pair and swap only when left>right. The largest remaining value moves to the final unsorted position after one left-to-right pass. Shrink the next pass by one position. Quadratic worst/average work comes from up to n(n-1)/2 pair comparisons.

Question sources

Data Structures & Algorithms review questions · Question 22

Question 240 | NS0541

In Selection Sort, what does each pass through the unsorted subarray accomplish?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Select the remaining minimum and put it at the next prefix position.

DSA33 scans the unsorted suffix for its minimum and exchanges it with A[i]. After each meaningful pass, the sorted prefix grows by one and contains the smallest keys placed so far. This differs from inserting the next encountered item into a prefix.

Primary note proof

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i]. There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order. At most n-1 exchanges of distinct positions are needed. A loop that also swaps the last item with itself can count n swap calls.

Question sources

Data Structures & Algorithms review questions · Question 23

Question 241 | NS0542

How does Insertion Sort differ from Selection Sort conceptually?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Selection finds a suffix minimum; insertion places the next item into a sorted prefix.

DSA33 chooses the globally smallest remaining key for each next output position. DSA34 instead takes the next input item, shifts larger prefix values, and inserts it into their gap. Both grow sorted prefixes, but choose their next item differently.

Primary note proof

Study-note section DSA33

> At position i, scan the remaining suffix for its minimum, then exchange it with A[i]. There are n-1 meaningful placements and n(n-1)/2 comparisons regardless of initial order. At most n-1 exchanges of distinct positions are needed. A loop that also swaps the last item with itself can count n swap calls.

Study-note section DSA34

> Begin with the first element already sorted. For each remaining item, save its value, shift larger preceding values right, and insert it into the gap. That gives n-1 insertion passes. On sorted input each pass needs one ordering check and no shifts: linear time. Reverse order causes 1+2+...+(n-1) shifts: quadratic time.

Question sources

Data Structures & Algorithms review questions · Question 24

Question 242 | NS0543

What is the best-case scenario for Bubble Sort and what is its complexity in that case?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Already sorted input; Θ(n) with no-swap early stopping.

DSA32 performs one pass over an already sorted array, makes no swaps, and stops when early-exit detection is enabled. That is linear best work. A fixed-pass implementation still executes its quadratic comparison schedule even on the same input.

Primary note proof

Study-note section DSA32

> Early stopping makes a sorted input linear: one no-swap pass finishes. A fixed pass schedule stays quadratic. Strict swaps preserve equal-key order. Cocktail sort runs bubble passes in both directions and is stable with strict swaps.

Question sources

Data Structures & Algorithms review questions · Question 25

Question 243 | NS0544

What are the two components of each node in a singly linked list?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Payload/data and a next-node link.

DSA08 defines a singly linked node as data plus next. The head reference points to the first node, and a null next terminates an ordinary noncircular chain. Head is a list-level reference rather than a second payload field inside every node.

Primary note proof

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Question sources

Data Structures & Algorithms review questions · Question 26

Question 244 | NS0545

How does a circular linked list differ from a singly linked list?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: A circular tail links back to head instead of ending at null.

DSA10 defines circularity by the tail-to-head link and stops traversal after returning to the start. This is independent of singly versus doubly linked: a singly linked list can itself be circular. Compared with the ordinary noncircular singly linked chain, termination changes from null to return-to-start.

Primary note proof

Study-note section DSA10

> Doubly linked nodes add prev; head and tail enable constant-time operations at both ends. Singly linked tail deletion still needs its predecessor. A circular tail links back to head; stop traversal on returning to the start, handling empty separately.

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Question sources

Data Structures & Algorithms review questions · Question 27

Question 245 | NS0546

To delete a node from the front of a singly linked list, what single update is needed?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Advance head to the old head's next link, then reclaim the old node.

DSA09 saves old=head, sets head=head.next, and releases old in an owned C++ chain. The essential link update is advancing head; check empty before dereferencing, and clear tail too if the only node was removed. This answers the operation without hiding ownership work.

Primary note proof

Study-note section DSA09

> Prepend with new.next=head; head=new. To insert after a known predecessor p, set new.next=p.next before p.next=new. Appending with only head requires a tail traversal; maintaining tail avoids it. Delete the head by saving old=head, advancing head=head.next and releasing old in C++.

Study-note section DSA09

> Deleting the tail of a singly linked list requires its predecessor, even when tail itself is known. Handle empty and single-node lists separately; after removing the only node, set both head and tail to null. To remove a known successor, bypass it before deallocation. Clearing an owned n-node chain visits and releases every node.

Question sources

Data Structures & Algorithms review questions · Question 28

Question 246 | NS0547

What extra pointer does each node in a doubly linked list have compared to a singly linked list?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: A prev link to the predecessor.

DSA10 says doubly linked nodes add prev alongside next. That extra link permits backward traversal and supports endpoint updates when head and tail are maintained. It does not automatically give constant arbitrary positional access, which may still need traversal.

Primary note proof

Study-note section DSA10

> Doubly linked nodes add prev; head and tail enable constant-time operations at both ends. Singly linked tail deletion still needs its predecessor. A circular tail links back to head; stop traversal on returning to the start, handling empty separately.

Question sources

Data Structures & Algorithms review questions · Question 29

Question 247 | NS0548

Why do linked lists have slower access times than arrays?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Arrays compute addresses directly; linked positional access follows a chain.

DSA02 computes an element address from base, index, and element size in constant time. DSA08's chain lacks direct index arithmetic, so reaching an arbitrary position may traverse linearly many links. This comparison concerns positional access rather than every operation: a known-head update can be constant.

Primary note proof

Study-note section DSA02

> For equal-size elements, address(A[i]) = base + i × element_size under zero-based indexing. This arithmetic gives constant-time positional access in the usual RAM cost model. A row-major rectangular matrix uses base + (row × columns + column) × element_size.

Study-note section DSA08

> A singly linked node contains data and next. Head points to the first node; null ends a noncircular chain. Empty means head==null. Nodes allocated as needed grow dynamically, with pointer overhead and no direct index arithmetic.

Question sources

Data Structures & Algorithms review questions · Question 30

Question 248 | NS0549

Convert the infix expression A + B * C - D / E to postfix.

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: ABC*+DE/-

DSA13 works the exact expression. Multiplication groups B*C and division groups D/E before addition and subtraction. Emit A, then B C *, then +, then D E /, then −, producing ABC*+DE/-.

Primary note proof

Study-note section DSA13

> Infix places an operator between operands and uses precedence and parentheses. Prefix places it first; postfix places it last. Their fixed operator arity determines grouping. To convert infix to postfix, output operands immediately and hold operators on a stack. Pop higher-precedence operators before pushing a new operator; also pop equal precedence for left-associative operators. Parentheses temporarily override this order.

Study-note section DSA13

> A+B*C-D/E becomes ABC*+DE/-.

Question sources

Data Structures & Algorithms review questions · Question 31

Question 249 | NS0550

What condition causes stack overflow in an array-implemented stack?

Hint

Use the cited definitions and invariants to distinguish the requested concept from related operations.

Answer: Attempting push when the fixed array is full.

DSA11 distinguishes full storage from the overflow attempt. Under zero-based top-as-last-occupied convention, full means top=C−1; a count convention uses size=C. A push must check this before writing beyond capacity.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA11

> Overflow is an attempted insertion into full storage; underflow is an attempted removal from empty storage.

Question sources

Data Structures & Algorithms review questions · Question 32

Question 250 | NS0551

What are the three main operations of a Stack and their time complexities?

Hint

State the representation and cost model, then count the operations or storage demanded by the question.

Answer: Push, pop, and peek are O(1) for fixed-array or head-linked stacks.

DSA11 defines push insertion, pop removal/return, and peek observation at top. Fixed-array indexing and linked-head updates take constant work. A resizing append or singly linked tail-pop implementation can change the cost, so the guarantee needs the stated representations.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA11

> A count-based alternative uses size=0 and A[size++]=x. Do not combine its empty/full tests with the top-index convention. Fixed-array push/pop are constant time. Linked stacks achieve constant time by inserting and deleting at the head; using a singly linked tail for pop would require a predecessor traversal.

Question sources

Data Structures & Algorithms review questions · Question 33

Question 251 | NS0552

In postfix notation, why are parentheses unnecessary?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Operator position and known arity determine grouping.

DSA13 places postfix operators after their operands. During stack evaluation, each binary operator consumes the two latest operand values in right-then-left order and pushes their result. The sequence therefore encodes grouping without precedence parentheses.

Primary note proof

Study-note section DSA13

> Infix places an operator between operands and uses precedence and parentheses. Prefix places it first; postfix places it last. Their fixed operator arity determines grouping. To convert infix to postfix, output operands immediately and hold operators on a stack. Pop higher-precedence operators before pushing a new operator; also pop equal precedence for left-associative operators. Parentheses temporarily override this order.

Study-note section DSA13

> To evaluate postfix, push operands. At a binary operator, pop right operand first, then left, and push left op right. Build an expression tree for either prefix (preorder) or postfix (postorder); a stack is useful but is not the only possible conversion representation.

Question sources

Data Structures & Algorithms review questions · Question 34

Question 252 | NS0553

What is the difference between a simple array queue and a circular queue?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Circular wraparound reuses freed front slots; a simple noncircular queue may not.

DSA15 distinguishes a rear index reaching the array end from circular modulo arithmetic. A circular queue can insert into positions freed at the front, while a simple noncircular version may report overflow despite those gaps. Capacity and empty/full conventions still must be consistent.

Primary note proof

Study-note section DSA15

> A simple noncircular array queue may report overflow when rear reaches the final index despite released slots at the front. A circular queue reuses slots with modulo arithmetic. Use a count convention: front points to the next removal, size counts items, and insertion slot is (front+size)%C. Empty means size==0; full means size==C.

Question sources

Data Structures & Algorithms review questions · Question 35

Question 253 | NS0554

A stable priority queue serves smaller numeric priority keys first and compares equal priorities by an increasing arrival counter, also smallest first. If two tasks have equal priority, in what order are they served?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: FIFO among equal priorities: the task that arrived earlier is served first.

The priority comparison ties, so the specified comparator proceeds to the arrival counter. An increasing counter gives the earlier arrival the smaller secondary key, causing it to be removed first. Equal-priority tasks are therefore served FIFO under this stable policy. That result follows from the explicitly chosen tie rule; an ordinary priority queue without a stability or comparator rule would not inherently guarantee arrival-order service.

Primary note proof

Study-note section DSA17

> Arrays and lists can also implement priority queues: an unordered sequence gives cheap insertion but linear extremum removal; an ordered sequence reverses that tradeoff. Equal priorities have no inherent FIFO guarantee. If stable service is required, compare (priority, arrival_counter). Choose the priority direction and tie rule before tracing.

Question sources

Data Structures & Algorithms review questions · Question 36

Question 254 | NS0555

What causes queue underflow and what does it indicate?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Dequeue on an empty queue causes underflow.

DSA11 defines underflow as attempted removal from empty storage; DSA15's count convention detects empty using size==0. This indicates that no item exists to return, not that the array is full or that an ordinary emptiness test itself failed.

Primary note proof

Study-note section DSA11

> Overflow is an attempted insertion into full storage; underflow is an attempted removal from empty storage. Merely checking isEmpty does not underflow.

Study-note section DSA15

> A simple noncircular array queue may report overflow when rear reaches the final index despite released slots at the front. A circular queue reuses slots with modulo arithmetic. Use a count convention: front points to the next removal, size counts items, and insertion slot is (front+size)%C. Empty means size==0; full means size==C.

Question sources

Data Structures & Algorithms review questions · Question 37

Question 255 | NS0556

Name three real-world applications of stacks.

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Runtime call frames, undo/redo history, and string or sequence reversal.

DSA12 gives nested calls latest-first returns, moving latest actions between undo and redo stacks, and reversing input by pushing then popping. Each example uses the same latest-pending-item discipline. A new edit after undo normally clears redo history.

Primary note proof

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Question sources

Data Structures & Algorithms review questions · Question 38

Question 256 | NS0557

What property must hold at every node of a Binary Search Tree?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Every left-subtree key is smaller and every right-subtree key larger, for distinct keys.

DSA19 imposes this invariant over entire subtrees at every node, not only immediate children. A descendant can satisfy its parent comparison yet violate an ancestor bound. Duplicate keys require an explicit count or placement policy instead of silently weakening the distinct-key rule.

Primary note proof

Study-note section DSA19

> With distinct keys, every key in a node's left subtree is smaller and every key in its right subtree is larger. Immediate-child comparisons alone are insufficient: root 10, left child 5, and 5's right child 12 violate the root's restriction. Duplicates require a separate count or placement policy.

Question sources

Data Structures & Algorithms review questions · Question 39

Question 257 | NS0558

List the order of nodes visited in Inorder traversal of BST: root=4, left subtree has 1,2,3, right has 5,6,7.

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: 1,2,3,4,5,6,7

DSA20 visits left subtree, node, then right subtree for inorder, and says BST inorder returns sorted keys. The supplied seven distinct keys therefore appear in increasing order even though the internal child shapes are not fully specified. Shape would matter to preorder or postorder.

Primary note proof

Study-note section DSA20

> Preorder visits node,left,right; inorder left,node,right; postorder left,right,node. Apply recursively; null children do nothing. Inorder returns sorted keys only for a BST. Preorder supports copying, postorder supports deleting children before parents, and level order uses a FIFO queue.

Study-note section DSA19

> With distinct keys, every key in a node's left subtree is smaller and every key in its right subtree is larger. Immediate-child comparisons alone are insufficient: root 10, left child 5, and 5's right child 12 violate the root's restriction. Duplicates require a separate count or placement policy.

Question sources

Data Structures & Algorithms review questions · Question 40

Question 258 | NS0559

When deleting a BST node with two children, what are the two valid replacement choices?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Inorder predecessor or inorder successor.

DSA21 identifies the maximum key in the left subtree and minimum key in the right subtree as valid replacement values. Copy one into the deleted node, then remove that value's original node, which has at most one child. These preserve the subtree order invariant.

Primary note proof

Study-note section DSA21

> A leaf can be detached by clearing its parent link. For one child, make the parent bypass the node and point to the child. For two children, copy the inorder predecessor (left-subtree maximum) or successor (right-subtree minimum), then delete that replacement from its original location. That second deletion has at most one child.

Question sources

Data Structures & Algorithms review questions · Question 41

Question 259 | NS0560

What is the time complexity of BST search in the best case vs worst case?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Best successful search Θ(1); arbitrary BST worst Θ(n), balanced worst Θ(log n).

DSA19 gives a constant root hit and O(height) search. A chain has linear height, producing linear worst work; logarithmic height disciplines in DSA22 improve that worst bound. Calling every BST logarithmic would incorrectly assume balance.

Primary note proof

Study-note section DSA19

> Search compares at each node and follows one child until equality or null. Insertion uses that same path to attach a new leaf. Operations cost O(height): logarithmic for balanced trees, linear for a chain. Best-case successful search is Theta(1), a root hit. An optimal BST minimizes expected search cost using access probabilities; constructing it is a different problem from searching it.

Study-note section DSA22

> AVL balance factor is height(left)-height(right); every node must have factor -1,0 or 1. After insertion or deletion, update heights upward and rotate where necessary. LL imbalance needs a right rotation; RR needs left; LR rotates the left child left then the node right; RL is symmetric. Rotations preserve inorder key order.

Question sources

Data Structures & Algorithms review questions · Question 42

Question 260 | NS0561

What is the key difference between BFS and DFS graph traversal?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: BFS explores distance layers with FIFO; DFS finishes branches with a stack or recursion.

DSA41 enqueues discoveries in layer order, while DSA42 resumes the latest unfinished branch before later neighbors. Both can traverse all reachable vertices, but their pending-work discipline and visitation order differ. BFS also has the unweighted minimum-edge guarantee.

Primary note proof

Study-note section DSA41

> Mark the source when enqueuing it. Repeatedly dequeue a vertex, examine its neighbours, and enqueue each unvisited neighbour while marking it. Store predecessor[v] when discovering v if a path must be reconstructed. FIFO ensures vertices at distance d are explored before later distance layers.

Study-note section DSA42

> Mark a vertex, then recursively visit each unmarked neighbour before continuing with later neighbours. An explicit stack can store pending work instead. Both implement latest-pending-work first; a queue would change that discipline. Marking prevents infinite repetition around graph cycles.

Question sources

Data Structures & Algorithms review questions · Question 43

Question 261 | NS0562

A directed graph has edges with direction; an undirected graph does not. Give one real-world example of each.

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Directed: following accounts; undirected: a mutual friendship relationship.

DSA40 gives following as directed because u following v need not imply v following u. It gives mutual friendship as undirected, representing a symmetric connection. The examples concern relation semantics rather than whether lines happen to be drawn with arrows.

Primary note proof

Study-note section DSA40

> A graph has vertices V and edges E. Directed edges distinguish u→v from v→u; undirected edges represent two-way relationships. Following accounts is directed; a mutual friendship relation is undirected. Weighted edges carry costs; unweighted shortest-path questions count edges. DAG expands to directed acyclic graph. Trees are connected acyclic graphs, although graph data structures may also contain cycles.

Question sources

Data Structures & Algorithms review questions · Question 44

Question 262 | NS0563

Why is a dense graph more computationally expensive to process than a sparse graph?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Edge-processing algorithms do more work when E is larger.

DSA40 says dense simple graphs approach quadratic edge counts and adjacency lists use Θ(V+E) space. Traversals or algorithms inspecting every edge therefore require more work as density increases. This is not universal: a vertex-only operation need not inspect edges.

Primary note proof

Study-note section DSA40

> A matrix uses Theta(V^2) space and constant edge lookup. Adjacency lists use Theta(V+E) space and enumerate neighbours. Dense simple graphs approach quadratic edge counts. Examining every edge costs more on denser graphs; vertex-only operations need not inspect every edge.

Study-note section DSA41

> BFS finds minimum-edge paths in unweighted graphs, not arbitrary minimum-weight paths. With adjacency lists it costs O(V+E); scanning a matrix costs O(V^2). To traverse a disconnected graph, restart from each unvisited vertex. Neighbour order affects tied traversal order but not minimum distances.

Question sources

Data Structures & Algorithms review questions · Question 45

Question 263 | NS0564

In a full binary tree, every node has either 0 or 2 children. How does this differ from a complete binary tree?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Full restricts each node to zero or two children; complete restricts level filling.

DSA18 defines full by each node's child count and complete by filled earlier levels plus left-filled final level. Those are distinct conditions: completeness can allow one child at the boundary, while fullness alone need not keep every level filled.

Primary note proof

Study-note section DSA18

> Binary means at most two children. Full means each node has zero or two children. Complete means all earlier levels are filled and the last fills from left. Perfect means all leaves share a level and every internal node has two children. Balanced means height stays logarithmic under a stated rule; it does not require a perfect shape.

Question sources

Data Structures & Algorithms review questions · Question 46

Question 264 | NS0565

What distinguishes a weighted graph from an unweighted graph, and when would you prefer weighted?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Weighted edges carry differing costs; use them when edge costs matter.

DSA40 assigns costs to weighted edges and says unweighted shortest-path questions count edges. Distances or travel costs can therefore justify weights when fewer edges is not the true objective. DSA43 then distinguishes weighted shortest paths from unweighted BFS.

Primary note proof

Study-note section DSA40

> A graph has vertices V and edges E. Directed edges distinguish u→v from v→u; undirected edges represent two-way relationships. Following accounts is directed; a mutual friendship relation is undirected. Weighted edges carry costs; unweighted shortest-path questions count edges. DAG expands to directed acyclic graph. Trees are connected acyclic graphs, although graph data structures may also contain cycles.

Study-note section DSA43

> For nonnegative weights, standard Dijkstra initializes source distance zero and others infinity, repeatedly finalizes the unsettled vertex with smallest distance, and relaxes each outgoing edge: dist[v]=min(dist[v],dist[u]+w(u,v)). A min-priority queue supports the next choice. BFS is enough when every edge has equal cost; negative edges require another justified method, such as Bellman-Ford.

Question sources

Data Structures & Algorithms review questions · Question 47

Question 265 | NS0566

If a BST receives keys in sorted order (1,2,3,4,5), what shape does the tree take and what is the search complexity?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: An ordinary unbalanced insertion forms a right chain; worst search is Θ(n).

DSA19 explicitly traces sorted insertion 1,2,3,4,5 into a right chain. Each new larger key follows right links to the tail, making the height linear. Worst search follows that chain, while a root hit remains constant. Balanced insertion would rotate and is a different assumption.

Primary note proof

Study-note section DSA19

> Search compares at each node and follows one child until equality or null. Insertion uses that same path to attach a new leaf. Operations cost O(height): logarithmic for balanced trees, linear for a chain. Best-case successful search is Theta(1), a root hit. An optimal BST minimizes expected search cost using access probabilities; constructing it is a different problem from searching it.

Study-note section DSA19

> Sorted insertions 1,2,3,4,5 instead form a right chain.

Question sources

Data Structures & Algorithms review questions · Question 48

Question 266 | NS0567

When would you choose a linked list over an array? Give a specific scenario.

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Choose links for frequent known-location updates, such as a doubly linked history cursor.

DSA10 gives browser history with a doubly linked cursor as an application. With references to current insertion or removal positions, link changes avoid shifting an array's later elements; arbitrary positional lookup can still be linear. This concrete scenario supports the choice without claiming universal superiority.

Primary note proof

Study-note section DSA10

> Lists suit changing sequences when references to insertion positions are already available. Finding a position is still linear. Lists can hold polynomial terms (coefficient, exponent), directory or block links, child lists in nonbinary trees, and hash chains. Browser history can use a doubly linked cursor. A playlist may be circular; airline seat lookup requires a specified seat-record model, not an automatic linked-list answer.

Study-note section DSA02

> A fixed array has a chosen capacity; the choice need not occur at compile time. A resizable array allocates a larger block and copies when needed. Inserting in the middle shifts later elements, so it costs linear time. Appending into spare capacity is constant; a resizing append can be linear, though repeated geometric growth gives amortized constant append cost.

Question sources

Data Structures & Algorithms review questions · Question 1

Question 267 | NS0568

How does a stack differ from a queue? Give an application for each.

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Stack: LIFO for undo; queue: FIFO for equal-priority printer jobs.

DSA11 gives latest-first stack service and DSA12 applies it to undoing the latest edit. DSA14 gives arrival-order queues for equal-priority printing. Both ADTs may use arrays or nodes; their service order, not that representation, is the distinction.

Primary note proof

Study-note section DSA11

> Stack policy is LIFO, equivalently FILO. Push inserts, pop removes/returns top, and peek only observes. With zero-based occupied slots, top=-1 is empty and top=C-1 is full. Check before A[++top]=x or returning A[top--].

Study-note section DSA14

> FIFO suits equal-priority printer jobs, shared-resource waiting, asynchronous buffers and level-order exploration. A play sequence can also use a queue. Distinguish airline seat records from waiting requests; an application name alone does not force one representation.

Study-note section DSA12

> Undo moves the latest action from undo to redo; redo moves it back.

Question sources

Data Structures & Algorithms review questions · Question 2

Question 268 | NS0569

Why is a balanced BST (like AVL) preferred over a regular BST? Explain the worst-case scenario.

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Balance prevents linear-height chains and guarantees logarithmic worst search.

DSA22 maintains AVL balance factors through rotations that preserve inorder. DSA19 shows ordinary sorted insertion producing a chain, where a search can traverse all nodes. Maintaining logarithmic height prevents that worst degradation, though root-hit best time remains constant.

Primary note proof

Study-note section DSA22

> AVL balance factor is height(left)-height(right); every node must have factor -1,0 or 1. After insertion or deletion, update heights upward and rotate where necessary. LL imbalance needs a right rotation; RR needs left; LR rotates the left child left then the node right; RL is symmetric. Rotations preserve inorder key order.

Study-note section DSA19

> Search compares at each node and follows one child until equality or null. Insertion uses that same path to attach a new leaf. Operations cost O(height): logarithmic for balanced trees, linear for a chain. Best-case successful search is Theta(1), a root hit. An optimal BST minimizes expected search cost using access probabilities; constructing it is a different problem from searching it.

Question sources

Data Structures & Algorithms review questions · Question 3

Question 269 | NS0570

What is a hash collision? Name two methods to resolve collisions and compare them.

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: A collision maps different keys to one home address; compare chaining and open addressing.

DSA26 stores colliding entries in bucket lists for chaining, whereas open addressing probes other array slots. Chaining uses link or bucket overhead and expected O(1+α) work; probing keeps entries inside the table and requires careful load and deletion policies. Neither prevents all collisions.

Primary note proof

Study-note section DSA26

> Separate chaining stores colliding entries in a bucket list. Load alpha=n/m is average chain length; expected operations are O(1+alpha) under suitable distribution assumptions, with O(n) worst cases. Open addressing keeps entries inside the array and probes until a key or an available slot is found.

Study-note section DSA26

> Linear probing tries (h(k)+j)%m. For size 7 and keys 10,17,24, all home addresses are 3 and occupied positions become 3,4,5. Quadratic probing changes the step; double hashing uses h1(k)+j h2(k), modulo m, with a step relatively prime to m to cover all slots. Neither method guarantees no collisions. Deletion needs tombstones or reinsertion so searches do not stop prematurely.

Question sources

Data Structures & Algorithms review questions · Question 4

Question 270 | NS0571

What is the difference between BFS and DFS? Which finds the shortest path?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: BFS uses FIFO layers and finds shortest unweighted paths; DFS uses branch-first stack order.

DSA41 guarantees minimum-edge paths for BFS on unweighted graphs. DSA42 says DFS paths are reachability witnesses but need not be shortest. Differing edge weights require an appropriate weighted-path algorithm, such as Dijkstra under nonnegative weights.

Primary note proof

Study-note section DSA41

> BFS finds minimum-edge paths in unweighted graphs, not arbitrary minimum-weight paths. With adjacency lists it costs $O(V+E)$; scanning a matrix costs $O(V^2)$. To traverse a disconnected graph, restart from each unvisited vertex. Neighbour order affects tied traversal order but not minimum distances.

Study-note section DSA42

> Using the same graph and alphabetical neighbours, recursive DFS can discover A,B,D,C. Its path is a valid reachability witness but not necessarily shortest. Adjacency-list cost is $O(V+E)$, with $O(V)$ visited state and possible stack depth. DFS supports cycle and connectivity analysis; a directed cycle test distinguishes active recursion vertices from already completed ones.

Study-note section DSA43

> BFS is enough when every edge has equal cost; negative edges require another justified method, such as Bellman-Ford.

Question sources

Data Structures & Algorithms review questions · Question 5

Algorithm Design & Analysis review questions · Question 4

Question 271 | NS0572

A text editor needs to implement multi-level undo. Which data structure is most appropriate and why?

Hint

Use the cited invariant or operation rule, state its assumptions, and trace a small case to justify the result.

Answer: Use an undo stack, with a second redo stack if needed.

DSA12 moves the latest action from undo to redo and reverses that movement for redo. Pushing each new edit and popping for undo naturally matches the most-recent-edit-first requirement. A new edit after undo normally clears redo history.

Primary note proof

Study-note section DSA12

> Calls nest on a runtime stack; the latest call returns first. Bracket matching pushes opening brackets and matches each closing bracket against top. Reversal pushes input then pops it. Undo moves the latest action from undo to redo; redo moves it back. A new edit after undo normally clears redo. DFS and backtracking use the same latest-pending-choice discipline.

Question sources

Data Structures & Algorithms review questions · Question 6